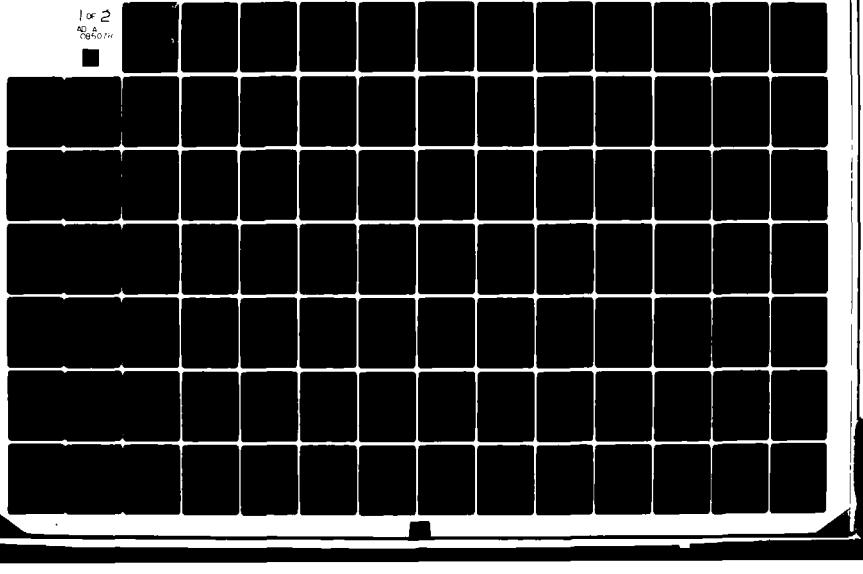
AD-A085 078 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
TEST GENERATION FOR MICROPROCESSORS.(U)
MAY 79 S M THATTE N00014-79-C-0424
UNCLASSIFIED R-842 NL



REPORT R-842 MAY, 1979 UILU-ENG 78-2235

CSL COORDINATED SCIENCE LABORATORY

LEVEL II

AD A085078

TEST GENERATION FOR MICROPROCESSORS

SATISH MUKUND THATTE

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A085 078
3. RECIPIENT'S CATALOG NUMBER (9) Doctoral thesis	
4. TITLE (and Subtitle) TEST GENERATION FOR MICROPROCESSORS	5. TYPE OF REPORT & PERIOD COVERED Technical Report
(14)	6. PERFORMING ORG. REPORT NUMBER R-842; UILU-ENG-78-2235
7. AUTHOR(s) (10) Satish Mukund Thatte	8. CONTRACT OR GRANT NUMBER(s) DAAB-07-72-C-0259; DAAG-29-78-C-0016 N00014-79-C-0424
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program (11)	12. REPORT DATE May 79
	13. NUMBER OF PAGES 167
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 174	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(15) N00014-77-C-0418; DAAE07-72-C-0259

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microprocessor Architecture
Architecture Models
Functional Level Fault Models
Test Programs
Complexity of Tests

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of this report is to develop test generation procedures for testing microprocessors in a user environment. Classical fault detection methods based on the gate and flip-flop level or on the state diagram level description of microprocessors are not suitable for test generation. The problem is further compounded by availability of a large variety of microprocessors. They differ widely in their organization, instruction repertoire, addressing modes, data storage and manipulation facilities, etc. In this report, a general graph-theoretic model for microprocessors is developed at the register transfer

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

level. Any microprocessor can be easily modeled using information only about the instruction set and the functions performed by it. This information is easily available in the user's manual. A fault model is developed on a functional level quite independent of the implementation details. The effects of faults in the fault model are investigated at the level of the graph-theoretic model. Test generation procedures are proposed which take the microprocessor organization and the instruction set as parameters and generate tests to detect all the faults in the fault model. The complexity of the test sequences measured in terms of the number of instructions is given. Our effort in generating tests for a real microprocessor and evaluating their fault coverage is described.

UILU-ENG 78-2235

TEST GENERATION FOR MICROPROCESSORS

by

Satish Mukund Thatte

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259, DAAG-29-78-C-0016 and NOOO14-79-C-0424.

TEST GENERATION FOR MICROPROCESSORS

Satish Mukund Thatte, Ph.D.
Coordinated Science Laboratory and
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1979

The goal of this thesis is to develop test generation procedures for testing microprocessors in a user environment. Classical fault detection methods based on the gate and flip-flop level or on the state diagram level description of microprocessors are not suitable for test generation. The problem is further compounded by availability of a large variety of microprocessors. They differ widely in their organization, instruction repertoire, addressing modes, data storage and manipulation facilities, etc. In this thesis, a general graph-theoretic model for microprocessors is developed at the register transfer level. Any microprocessor can be easily modeled using information only about the instruction set and the functions performed by it. This information is easily available in the user's manual. A fault model is developed on a functional level quite independent of the implementation details. The effects of faults in the fault model are investigated at the level of the graph-theoretic model. Test generation procedures are proposed which take the microprocessor organization and the instruction set as parameters and generate tests to detect all the faults in the fault model. The complexity of the test sequences measured in terms of the number of instructions is given. Our effort in generating tests for a real microprocessor and evaluating their fault coverage is described.

TEST GENERATION FOR MICROPROCESSORS

BY

SATISH MUKUND THATTE

B.E. (Hons.), Birla Institute of Technology and Science, 1975
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1979

Thesis Advisor: Professor Jacob A. Abraham

Urbana, Illinois

ACKNOWLEDGEMENT

I wish to express my gratitude to Professor Jacob Abraham for his guidance and supervision throughout my graduate work. I would also like to thank Professor Gernot Metze and Professor Edward Davidson for their encouragement and guidance.

I appreciate the help rendered by Dr. Ken Parker of the Hewlett-Packard Company, Loveland Instrument Division, Loveland, Colorado. The TESTAID III fault simulator and the gate level description of the microprocessor provided by Dr. Parker were indispensable in conducting our case study consisting of the generation of test sequences for a real microprocessor and the evaluation of fault coverage. I would also like to gratefully acknowledge the assistance provided by Dick Norton and Tom Lovett in this case study.

I would like to thank Ravi Nair (now with IBM) for many useful discussions contributing to this thesis. I am also thankful to all my colleagues at the Coordinated Science Laboratory for providing an intellectually stimulating and enjoyable environment. Finally, special thanks are due to Mrs. Gertrude Little for typing this thesis.

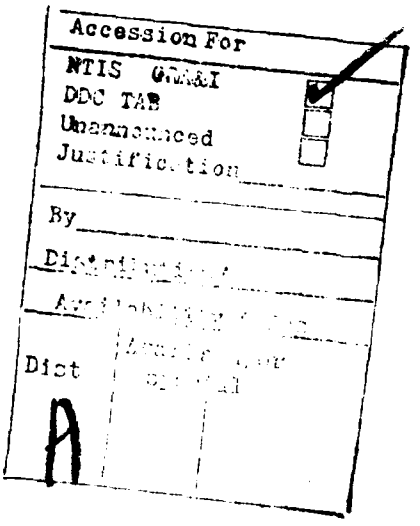


TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1. Description of the Problem

Microprocessors constitute a very high percentage of today's large scale integrated (LSI) circuits. The number of microprocessor-based digital systems is expanding rapidly. This has given rise to an acute need for sound theoretical tools to develop efficient, thorough and cost-effective test programs to detect faults in microprocessors at all levels: at the component level during fabrication and before encapsulation, at the chip level before incorporating the microprocessor into a system, and at the system level in the field. These levels have their own testing requirements and constraints on test development and application.

Manufacturers of microprocessors are interested in testing various components and devices on the microprocessor chip during its fabrication for DC parametric behavior (such as power consumption, noise sensitivity, fanin and fanout capability, etc.) as well as dynamic timing problems, etc. Both manufacturers and users are interested in testing microprocessors at the chip level for its correct functional operation at the rated speed. Finally, users and system designers are interested in ensuring that the microprocessor in the system (as well as the rest of the system) is functioning correctly. Classical fault detection methods such as the D-algorithm [RBSc67] used for the chip and system level testing are employed to detect logical faults defined at a low level such as a line stuck-at-one or stuck-at-zero [CMMe70] and [BrFr76]. These faults are associated with lines interconnecting

gates and flip-flops. For microprocessors which contain thousands of gates, flip-flops and interconnections, classical methods must consider a very large number of faults making test generation extremely complicated.

Our approach associates faults with various functions of the microprocessor (defined at a suitably higher level), such as the register decoding function, instruction decoding and control function, data storage function, etc. We give some examples of faults in microprocessors which we are interested in detecting.

Example 1.1: When the instruction "Load register R_1" is executed, register R_2 is loaded instead of register R_1. This may happen due to a faulty register decoding function. The instruction "Interrupt enable" correctly enables the interrupt, but at the same time the accumulator is cleared. This can be attributed to a fault in the instruction decoding and control function. The instruction "Add the contents of register R_1 to the contents of the accumulator," is not correctly executed for a few operands due to faults in the arithmetic and logic (ALU) unit. We associate these faults with the data manipulation function. A register may fail to store certain data patterns. This fault is associated with the data storage function. □

Another important reason motivating our approach of considering faults at a functional level is the constraint imposed on testing microprocessors in a user environment: the test programs need to be generated without knowing the implementation details of the chip at the gate and flip-flop level. The only source of information which is readily

available is the typical user's manual detailing the instruction set and describing the architecture of the microprocessor. Using this information it is easier to define the functional behavior of a microprocessor and associate faults with the functions as illustrated in Example 1.1.

In this thesis we are concerned with formulating a sound theoretical foundation for test program generation for testing microprocessors in a user environment, particularly at the chip level. We are interested only in generating deterministic tests to detect permanent faults which give rise to faulty functional behavior as described in Example 1.1. We will not discuss testing issues related with dynamic timing problems, faulty DC parametric behavior or manufacturing or design processes. For these aspects readers are referred to [TEST75]. Of course, the "solution" that proposes the execution of each instruction for all possible operands and in every possible sequence for testing microprocessors is really not a solution. It only serves the purpose of dramatically pointing out how difficult the problem really is.

We assume that the external tester monitors all the input and output pins of the microprocessor. In particular, the status pins and the data and address buses of the microprocessor are continually checked. Testing is stopped on the detection of any fault, (may or may not be in real time) since we are not interested in fault location on a chip. The tester and the external memory which contains the instructions executed by the microprocessor are assumed to be fault free. Various sophisticated testers which are commercially available do satisfy the requirements mentioned above. In this thesis we will not discuss the design and implementation or operation of a tester. For information on testers

readers are referred to [Hust74] and [Ande76].

Sophisticated testers available for testing microprocessor chips cannot be conveniently used for testing microprocessors incorporated in a system in the field. Recognizing this difficulty various schemes such as self testing [Ba1179], [LiDo79] and transition counting [Haye76] have been proposed. A notable instrument suitable for field testing and diagnosis is the signature analyzer available from the Hewlett-Packard Company [HPJ077]. These techniques are aimed at the ability to test systems in the field without requiring a sophisticated tester; however, their fault detection capability principally hinges on how thorough the test programs are, again emphasizing the need for good test generation procedures. Though the test generation procedures proposed in this thesis assume the presence of a sophisticated tester, we believe that these procedures can be used, with relatively easy modifications, for generating tests suitable for field testing. However, more research is required in this area.

1.2. Thesis Outline

The goal of the thesis is to develop test generation procedures for testing microprocessors. These procedures should treat the microprocessor organization and instruction set as parameters. This is necessary in view of the fact that today's microprocessors differ widely in their organization, instruction repertoire, addressing modes, data storage and manipulation facilities, etc.

In Figure 1.1 the thesis outline is schematically illustrated. In the beginning of Chapter 2 we survey various models and methods of

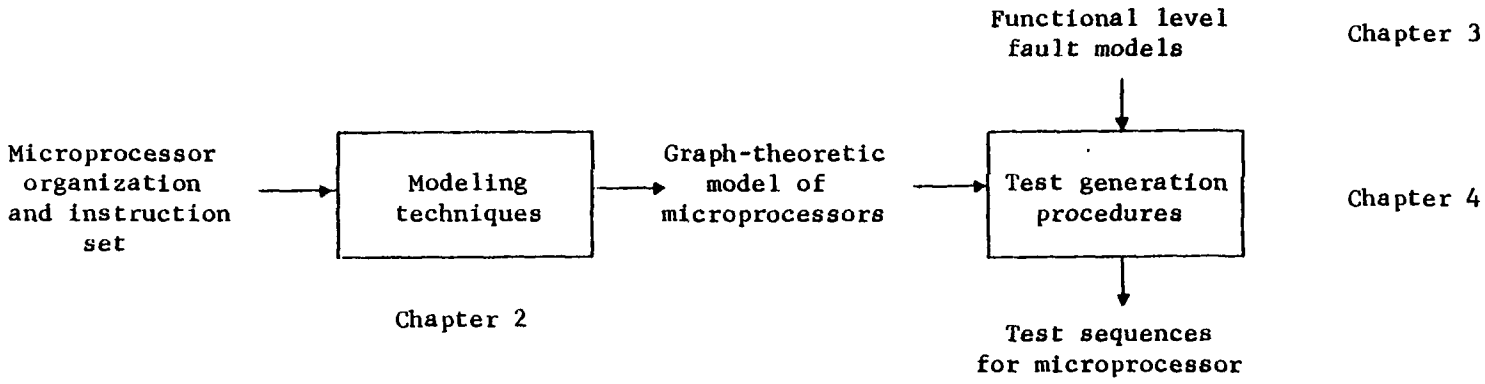


Figure 1.1. Illustrating the thesis outline.

test generation published in literature. Since none of them is suitable for testing microprocessors, we first develop a general graph-theoretic model for microprocessors at the register transfer level. Any microprocessor can be easily modeled on the proposed lines using information about the instruction set and the functions performed by it. (This information is easily available in a typical user's manual.) This leads us to treat the microprocessor organization and the instruction set as parameters of the test generation procedures. We will illustrate how to generate the graph-theoretic model for a small example microprocessor.

Functional level fault models capable of describing faulty behavior at a higher level are presented in Chapter 3. These models are quite independent of implementation details of a microprocessor. We will investigate the effects of these faults on the graph-theoretic model of a microprocessor. In Chapter 4 we will present test generation procedures to detect faults in the fault models and prove their fault coverage. The generation of the test sequence will be illustrated for the example microprocessor. The generated test sequences comprise valid machine instructions which are assembled to produce test patterns. This may be contrasted with the classical methods which may generate bit vectors that do not correspond to any instruction.

Chapter 5 discusses the complexity of the test sequences measured in terms of the number of instructions present in these sequences. Chapter 6 reports on the feasibility of our approach. We will describe our effort in generating tests for a real microprocessor. The results were quite encouraging. Finally, in Chapter 7, we summarize the thesis and suggest topics for future research.

2. A MODEL FOR MICROPROCESSORS

Any rigorous exercise of generating tests for fault detection in a digital system should consist of three activities:

1. Constructing a model at a suitable level for describing the behavior of the digital system.
2. Developing a fault model to define the scope of allowable faults in the system. A good fault model is usually found as a result of a trade-off between the need to account for most of the faults commonly observed in the system and the need to be able to keep the complexity of test generation low, and the length of tests short. The nature of the fault model is usually influenced by the model used to describe the system as illustrated in Section 2.1 below.
3. Generating tests to detect all the faults in the fault model.

Microprocessor testing practised in industry seems to be based on ad hoc techniques such as "testing" each instruction for many operands, "exercising" various modules in the microprocessor (such as the ALU, shifter, registers, indexing hardware), or running an application program. A typical example based on such ad hoc techniques is [ChMc76]. A good tutorial survey of testing methods and tools used in industry can be found in [FeeW78]. Other sources of information describing testing strategies practised in industry are the digests of the annual Semiconductor Test Symposiums sponsored by the IEEE Computer Society [TEST75]. These techniques are not based on a general model for microprocessors. Moreover, they do not consider any specific fault model. Therefore, the technique followed for testing one microprocessor may be difficult to extend to

other microprocessors having different architectures. It is also very difficult to know what faults can or cannot be detected using these techniques.

We now briefly review various models used in the literature for describing digital systems. We will comment on their applicability for modeling microprocessors for the purpose of test generation, particularly in a user environment.

2.1. Review of Previous Models

At the lowest level of the modeling spectrum, models are based on the gate and flip-flop level description of a digital system in order to describe its logic behavior. Most of the classical work on fault diagnosis uses these models. At the highest level of the spectrum, models are based on the so-called "black box" description of the system; truth tables are used to describe a combinational circuit and state tables are employed for describing a sequential circuit. As described below, both of these extremes are unsuitable for modeling microprocessors for the purpose of generating tests for them.

2.1.1. Gate and Flip-Flop Level Model

The system is described by a logic diagram consisting of gates and flip-flops. Thus gates and flip-flops are recognized as primitive elements. This model usually supports low-level fault models such as a line stuck-at-one or stuck-at-zero model, which associates faults with lines interconnecting gates and flip-flops [CMMe70] and [BrFr76]. These models were used to test and diagnose digital computers designed with discrete components and with a knowledge of the detailed logic description

[Mann66]. For an excellent annotated bibliography on this topic, readers are referred to [Scol72].

These models are not very useful for generating tests for LSI circuits such as microprocessors which contain a very large number of gates, flip-flops, and interconnections and which therefore require an enormous amount of computation to generate comprehensive test sets. In, addition, the required gate and flip-flop level description is usually not available to microprocessor test designers working in a user environment.

2.1.2. State Diagram Model

This model is based on the state diagram description of the system, giving its output and the next state for any input and present state. The system is viewed as a black box and all the implementation details are ignored. Several test methods have been proposed [Koha70] based on automata identification experiments. Though this model supports a very general fault model, the length of the test sequence generated grows exponentially with the number of inputs and states. This restricts the use of the method only to toy systems having a very small number of inputs and states and rules out its applicability to microprocessors.

2.1.3. Module Level Model

This model views a digital system as a network of interconnected modules such as the ALU, register file, multiplexers, demultiplexers, shifters, control unit, etc. Thus the primitive elements are these higher level modules instead of gates and flip-flops.

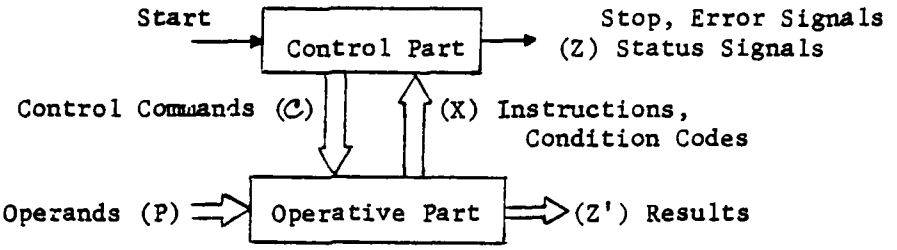
The problem is to generate tests for the entire system using the available tests for individual modules. This problem as stated above

is an extremely difficult and unsolved problem [Powe69] and [BaKi76], though in [BaKi76] methods are given to generate tests for purely combinational systems with some hardware modifications. This model is also not very promising at its present state of research for generating tests for microprocessors because microprocessors also contain sequential logic and no hardware modification is possible in an existing microprocessor chip.

2.1.4. The Robach-Saucier Model

In view of the difficulties pointed out in the previous sections, Robach and Saucier [RoSa75] and [RoSa78] proposed the following model for generating tests for control units of digital systems. Every system can be decomposed into two subsystems, the control and operative parts, as shown in Figure 2.1(a). The control part is characterized by a representation matrix M as shown in Figure 2.1(b). It has n rows corresponding to the set of elementary commands $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$, and m columns corresponding to the set of control states $\{Q_1, Q_2, \ldots, Q_m\}$, such that $m_{ij} = 1$, if the state Q_j activates the command c_i, and $m_{ij} = 0$ otherwise. The operative part can be considered to be made up of a set of independent functional units. The set of commands $\mathcal{C}$ is sent to one or more functional units.

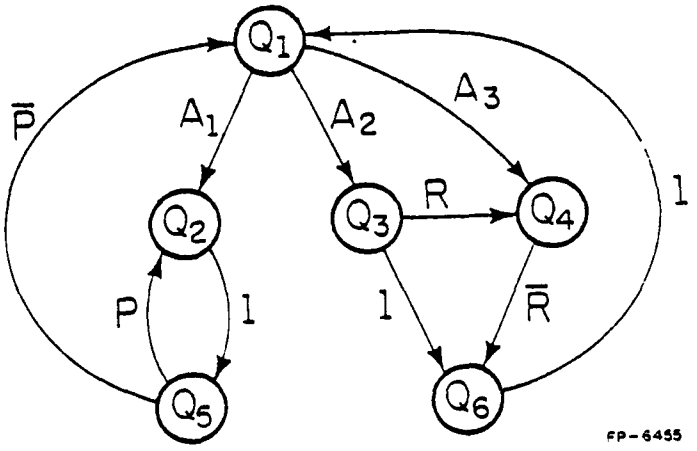
The diagnosis of the control part is performed through the operative part which is assumed fault free. The system is capable of performing a set of "algorithms." For these algorithms, the functioning of the control part is represented by a flow-chart as shown in Figure 2.1(c) where the nodes are the different control states of the considered



(a). A general model for a system.

	Q_1	Q_2	Q_3	.	.	.	Q_m
c_1	0	1	0	.	.	.	0
c_2	0	0	0	.	.	.	0
c_3	1	0	0	.	.	.	1
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
c_n	0	0	0	.	.	.	1

(b). Representation matrix M.



(c). Control representation for three algorithms A_1, A_2, and A_3

Figure 2.1. The Robach-Saucier Model.

algorithms, and the edges represent the different possible transitions between these states; the edges are labeled with the transition conditions (predicates).

A test of the control unit consists of complete identification of the states (distinguishability of every state from all other states, and checking the commands generated by each state through each functional unit), and verfication of sequences. The fault model allows commands to be abnormally activated or abnormally inactive. Thus this model considers basic control commands and control states as primitive elements rather than gates and flip-flops generating these commands.

This approach runs into two problems when applied for generating tests for microprocessors. First, the required information about the details of control states, basic control commands emitted during a state, and flow-charts for algorithms (i.e., each instruction of the microprocessor) may not be available to a test designer working in a user environment. This problem perhaps could be alleviated by formulating the control states and commands at a higher level. Even then, the method faces a second problem; as shown in Figure 2.1(a), it is assumed that the operands (denoted by P) required for the functional units are directly available, and the results (denoted by Z') produced by the functional units are directly observable (possibly with some time delay). On the other hand, in the case of microprocessors a sequence of instructions needs to be executed, in general, to provide proper operands to a functional unit (or to store data in a register) and to read out the result of an operation performed by it (or to read out the contents of a register). For example, consider an "Add" instruction which adds the contents of an accumulator and a

scratch-pad register and stores the result in the accumulator. A "Load accumulator" instruction is needed to load an operand in the accumulator, while two instructions may be required to store an operand in the scratch-pad register: the first to load a general purpose register and the second to transfer the contents from the register to the scratch-pad register. Similarly the accumulator can be read out only by executing the "Store accumulator" instruction.

Though the Robach-Saucier approach is a step in the right direction for testing certain digital systems where the assumptions made in their model are valid, it appears that the limited observability and controllability of internal registers and logic of microprocessors pose a very difficult problem in extending the approach to microprocessor testing.

2.1.5. The Thatte-Abraham Model

A methodology for test generation based on a model for a restricted but "typical" microprocessor organization and instruction set was proposed by Thatte and Abraham [ThAb78]. The model considers an organization for the data processing section of microprocessors shown in Figure 2.2 and allows only limited but commonly observed types of instructions, such as instructions performing information transfers between the main memory and level 1 registers, instructions performing various ALU operations, instructions performing information transfers among level 1 registers, between level 1 and 2 registers, and among level 2 registers.

The fault model takes into account faults associated with registers, ALU, buses, and control section such as incorrect decoding of instructions, missing and extraneous control commands, etc. A drawback

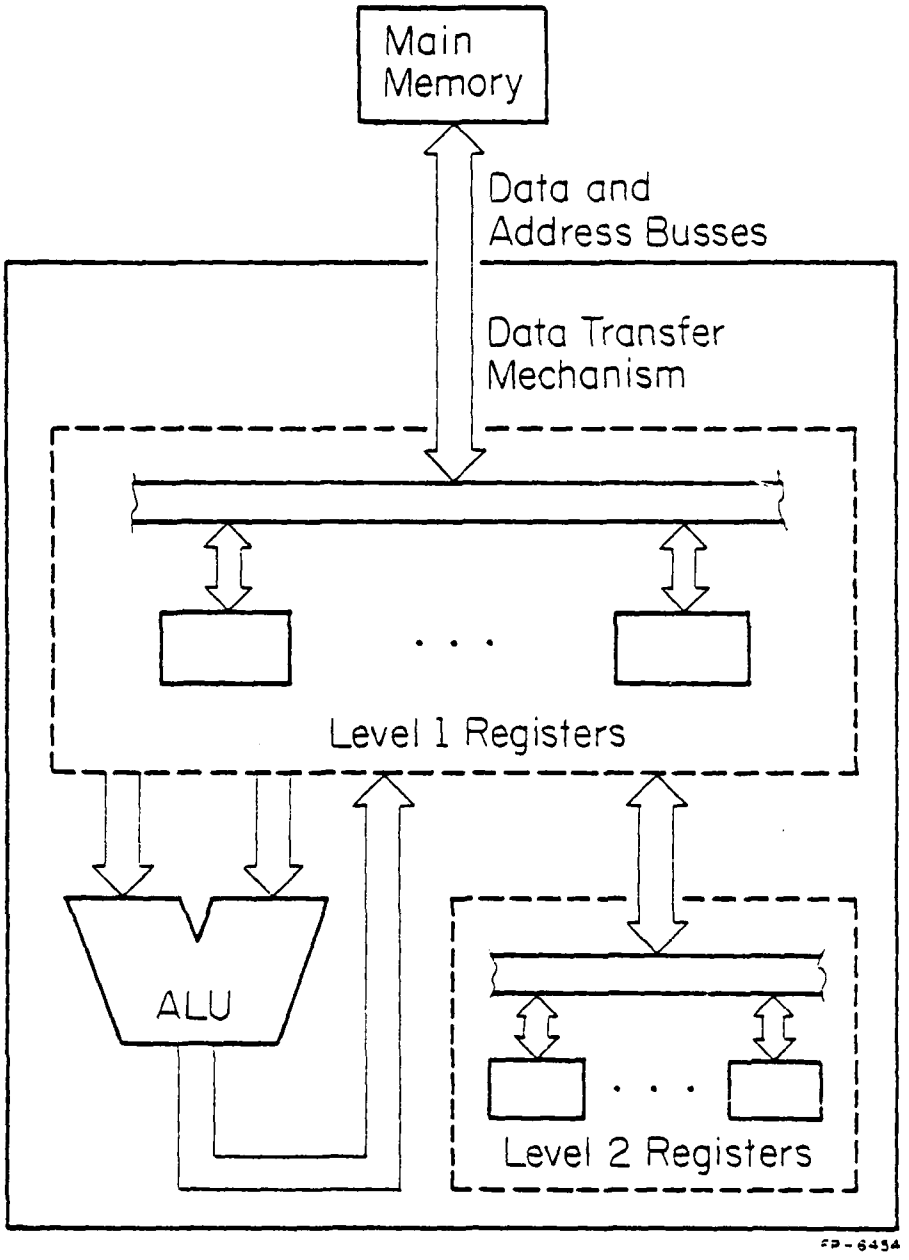


Figure 2.2. The Thatte-Abraham model.

of this model is that it cannot generate tests for different microprocessor organizations and instruction sets, i.e., it fails to treat the microprocessor architecture as a parameter of test generation.

2.2. A New Model for Microprocessors

In the light of the discussion in Sections 2.1.1 through 2.1.5, we summarize various requirements for a model suitable for generating tests for microprocessors.

1. The model should be based on a functional description defined at a suitably higher level such as the register transfer level. We will define the model in terms of data flow that occurs between various registers and the main memory during the execution of an instruction. This allows us to describe the functional behavior of a microprocessor by using information about the instruction set and functions performed by it. Since this information is readily available in a typical user's manual, this model is quite suitable, especially in a user environment.

2. The model should be able to treat the microprocessor organization and instruction set as parameters of the test generation procedure, so that for a given microprocessor architecture it would be possible to generate tests. This feature is very desirable as today's microprocessor differ widely in their organization and instruction set, addressing modes, etc. This trend is bound to continue with newly emerging and powerful microprocessors.

3. The model should be able to support a fault model describing faults in various functional primitives such as the data transfer function, the data storage function, the instruction decoding and control function, etc.,

allowing us to describe faulty behavior without knowing the details of their implementation.

For modeling purposes we partition the instruction repertoire of the microprocessor into three classes. This classification scheme is very similar to that proposed by Flynn [Flyn74].

1. Transfer class (denoted by class T):
Instructions of this class perform data transfer between the main memory and a register (on the microprocessor chip), between an I/O device and a register, between registers, and between the main memory locations. Examples are "Load accumulator," "Transfer register R_1 to register R_2," I/O instructions, etc.

2. Manipulation class (denoted by class M):
Instructions of this class manipulate the data stored in the main memory or registers by performing operations like "Shift," "Add," "Or," "Decrement," "Compare" instruction, etc.

3. Branch class denoted by class B):
Class B consists of all those instructions which do not belong to class T or M, e.g., "Conditional and unconditional branches," "Jump to subroutine" and "Return from subroutine (i.e., instructions associated with program sequencing)," "Interrupt enable and disable," "No operation" instruction, etc.

The proposed model is graph-theoretic in nature. Before defining the model formally, we motivate it by means of an example. We represent registers of the microprocessor by labeled nodes and instructions by

directed labeled edges where edges represent data* flow during the fetching and execution of instructions.

Example 2.1: Consider a simple register transfer instruction I_1 transferring the contents of register R_1 to register R_2. The data flow involved during the fetching and execution of I_1 can be represented in a graph as shown in Figure 2.3. Nodes R_3 and R_4 represent the instruction register and the program counter, respectively. The edge from node R_4 to OUT represents the transfer of address of a main memory location containing instruction I_1 from the program counter to the address register of the main memory. This edge is labeled I_1^1. The edge from node IN to R_3 represents the transfer of instruction I_1 from the main memory to the instruction register. This edge is labeled I_1^2. While the instruction transfer is taking place, the program counter is incremented. The self loop around node R_4 represents the function of incrementing the program counter. This loop is also labeled I_1^2. The edge from node R_1 to R_2 indicates the transfer of data stored in register R_1 to register R_2, i.e., the intended function of the instruction. This edge is labeled I_1^3. Notice that the data flow represented by the edge labeled I_1^1 takes place before that represented by the edges labeled I_1^2 which, in turn, takes place before that represented by the edge labeled I_1^3, and so on. Thus the superscripts on the edge labels indicate a precedence relation in time. □

*We use data as a generic term referring to the information as well as its address.

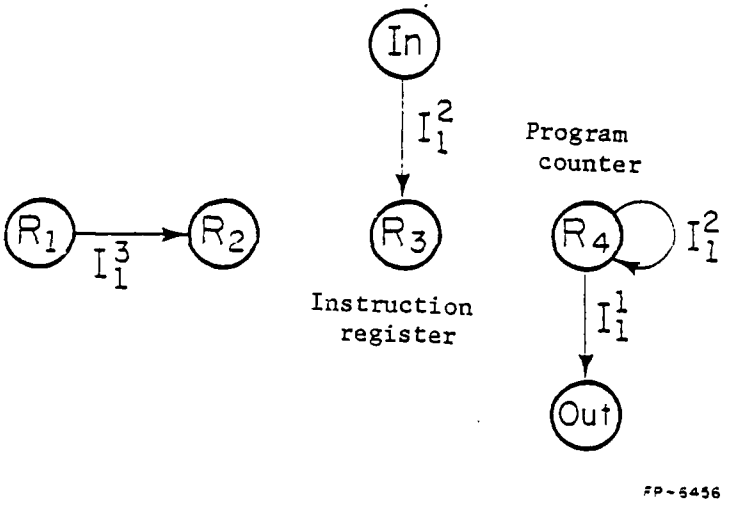


Figure 2.3. Representation of instruction I_1.

Example 2.1 points out some improvements that can lead to much more concise and succinct representation of an instruction: we may represent that data flow which is involved only during the execution of the instruction and not during its fetching. Since the data flow involved during the fetching of every instruction is the same, no information is gained by representing it for each instruction. On the other hand the data flow involved during the execution of an instruction really characterizes the function performed by the instruction. Thus only the edge from node R_1 to R_2 in Figure 2.3 can be used to represent instruction I_1.

We now formalize the model. A microprocessor is modeled by a system graph (S-graph). Let $\mathcal{R} = \{R_1, R_2, R_3, \ldots\}$ denote the set of registers in the microprocessor. Set $\mathcal{R}$ includes the so-called general purpose registers, accumulators, scratch-pad registers, on-chip last-in first-out stack, and the program counter. It also includes index registers, address buffer register, stack pointer, etc., i.e., the registers used in various addressing modes. Included also is the so-called processor status word containing various processor status bits. Each register R_i is represented by a node (labeled R_i) of the S-graph. In addition to the nodes representing registers, we incorporate two more nodes, named "IN" and "OUT" in the S-graph, representing the world external to the microprocessor, i.e., the main memory and I/O devices.

Let $\mathcal{I} = \{I_1, I_2, I_3, \ldots\}$ denote the set of instructions. The execution of instruction I_j causes data flow among a set of registers, and

between the main memory (or an I/O device) and registers in some sequence. The data may or may not be manipulated during the flow. We can represent the data flow during the execution of any instruction I_j as follows:

1. There exists a labeled directed edge from node R_p to node R_q, if data flow occurs from register R_p to register R_q (with or without manipulation) during the execution of I_j.

2. There exists a labeled directed edge from node IN to node R_j, if data flow occurs from the main memory or an I/O device to register R_j (with or without manipulation) during the execution of I_j.

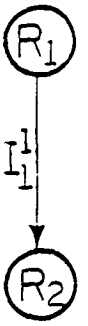
3. There exists a labeled directed edge from node R_j to node OUT, if data flow occurs from register R_j to the main memory (or its address register) or an I/O device (with or without manipulation) during the execution of I_j.

If more than one edge is required to represent the data flow during the execution of an instruction, the flow may occur in a specific sequence. The exact sequence may not be known to test designers working in a user environment because the sequence depends on the details of implementation of the microprocessor hardware. However, it is possible to deduce the precedence relation in time between the components of the data flow solely on the basis of logical data dependence, independent of the details of implementation. We indicate the precedence relation by means of the labels assigned to directed edges as explained below.

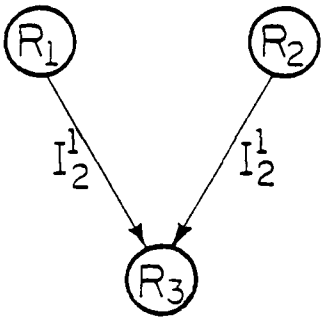
Among the set of edges representing the data flow during the execution of instruction I_j, two edges are labeled I_j^p and I_j^q, where $p < q$

and p and q are smallest such positive integers, if and only if the data flow represented by the edge labeled I_j^p must take place before that represented by the edge labeled I_j^q in order to preserve the underlying logical data dependence. Two edges are assigned the same label I_j^p, if and only if the corresponding data flows can occur simultaneously given the necessary number of resources such as buses and functional units, i.e., the required hardware parallelism exists. In the presence of some limitation on hardware resources, the data flow represented by these two edges may occur in either of the two possible sequences depending on the details of implementation. If only one edge is required to represent the data flow during the execution of instruction I_j, it is assigned a label I_j^1. It must be stated that this elaborate notation is used only for clarity in illustrating the data flow sequence and is not really necessary to generate tests.

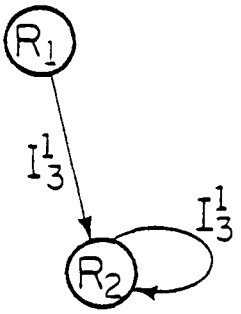
Example 2.2: Figure 2.4(a) represents a "Transfer" instruction I_1 that transfers the contents of register R_1 to register R_2. Figure 2.4(b) depicts an "Add" instruction I_2 which adds the contents of registers R_1 and R_2 and stores the result in R_3. Note that both edges are labeled I_2^1. If two separate buses are available to route the contents of registers R_1 and R_2 to the ALU simultaneously, the data flow represented by these two edges can take place in parallel. If only one bus is available, the contents of either R_1 or R_2 are transferred to the ALU first and stored in its latch, followed by the transfer of the contents of the other, and then the addition takes place. The actual implementation determines which register is selected first for data transfer.



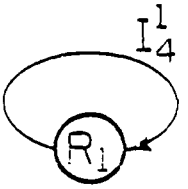
(a). I_1- Transfer instruction



(b). I_2- Add instruction



(c). I_3- Or instruction



(d). I_4- Rotate left instruction

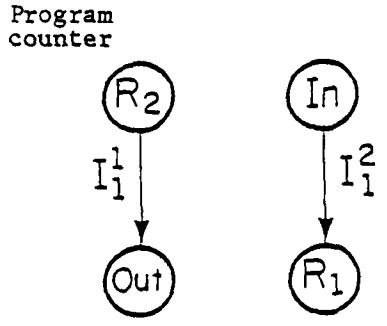
FP-6300

Figure 2.4. Representation of instructions.

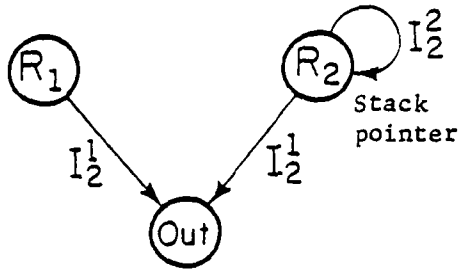
Figure 2.4(c) shows an "Or" instruction I_3 which forms the logical OR of the contents of registers R_1 and R_2 and stores the result in R_2. As explained above, both edges are labeled I_3^1. Figure 2.4(d) shows a "Rotate left" instruction which rotates the contents of register R_1 left by 1 bit. □

We now explain how to represent instructions which use addressing modes by these graph-theoretic techniques. A variety of addressing modes is usually available for instructions for fetching operands from the main memory and storing results into the main memory. Various examples of addressing modes are direct, indirect, immediate, indexed, relative, stack, etc. [GsMc75]. Each addressing mode is characterized by a sequence of data transfers between registers and the main memory.

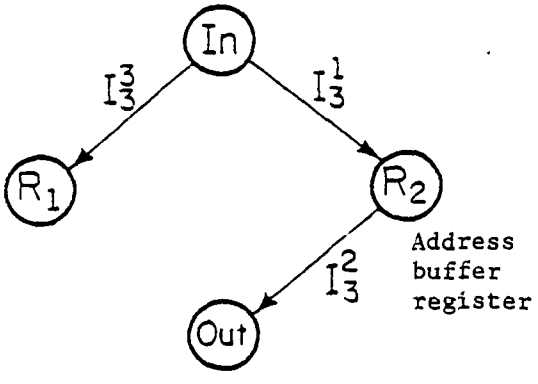
Example 2.3: Figure 2.5(a) represents the "Load register R_1" instruction, I_1, using the so-called implied or implicit addressing [GsMc75] where the data to be loaded is contained in the memory location next to the one storing instruction I_1, i.e., the address of the operand is derived by implication. The edge from node R_2 to OUT represents the transfer of the address of the operand from the program counter (R_2), (which is incremented by 1 by this time and points to the word next to the one storing instruction I_1), to the main memory address register, and the edge from IN to R_1 represents the data transfer from the main memory to register R_1. Figure 2.5(b) represents the "Stack push" instruction I_2 which pushes the contents of R_1 into the memory location (top of the last-in first-out (LIFO) stack maintained in the main memory) pointed to by the stack-pointer R_2 and then increments the stack pointer. The edge from node R_1 to node OUT in Figure 2.5(b) represents the transfer of data



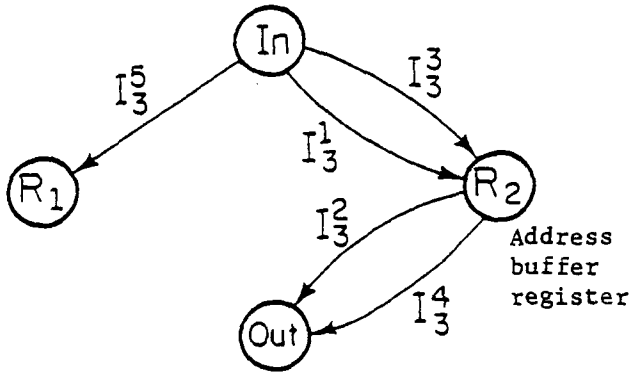
(a). I_1- Load register R_1 instruction using implied addressing



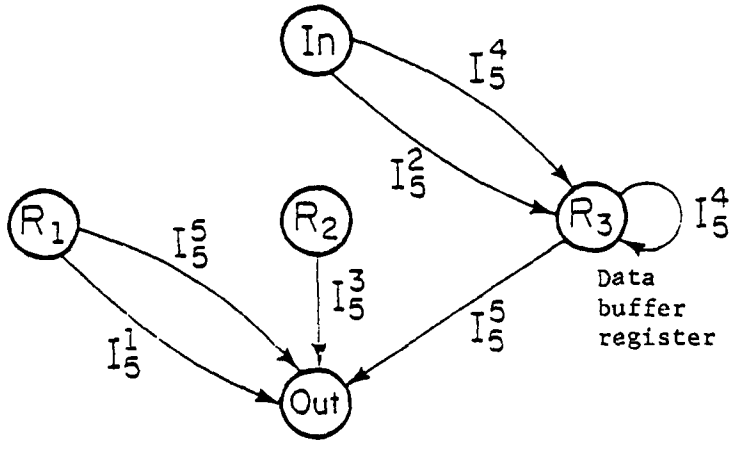
(b). I_2- Stack push instruction



(c). I_3- Load register R_1 instruction using direct addressing



(d). I_4- Load register R_1 instruction using indirect addressing



(e). I_5- Add $(R_1),(R_2)$ instruction

Figure 2.5. Representation of addressing modes.

(to be pushed on the stack) from R_1 to the LIFO stack. The R_2-OUT edge represents the transfer of the address of the top of the stack from R_2 (stack pointer) to the address register of the main memory. Both these edges are labeled I_2^1. The self loop around the node R_2 represents the stack pointer incrementing function. It is labeled I_2^2 because the stack pointer must be incremented only after the data is pushed so that it points to the location on the stack where next data can be pushed.

Figure 2.5(c) shows how to represent a register load instruction I_3 using direct addressing [GsMc75]. As shown in Figure 2.5(c) the address of the location storing the operand is fetched from the address field of instruction I_3 into the address buffer register R_2 (represented by the IN-R_2 edge labeled I_3^1). This address is then sent from R_2 to the address register of the main memory (represented by the R_2-OUT edge labeled I_3^2) and the operand is fetched from the main memory and loaded into R_1 (represented by the IN-R_1 edge labeled I_3^3). The register load instruction I_4 using indirect addressing mode can be represented by incorporating two more edges in Figure 2.5(c) (one more edge from node R_2 to OUT and one more edge from node IN to R_2) as shown in Figure 2.5(d).

Figure 2.5(e) shows the representation of a complicated instruction I_5 "Add (R_1), (R_2)," where the contents of registers R_1 and R_2 denote addresses of operands. The first operand for this instruction is fetched from the main memory location pointed to by register R_1 and stored in the data buffer register R_3 (accounting for the R_1-OUT edge labeled I_5^1 and the IN-R_3 edge labeled I_5^2). The second operand is fetched from the main memory location pointed to by register R_2 and it is added to the contents of register R_3 (accounting for the R_2-OUT edge labeled I_5^3, the other edge

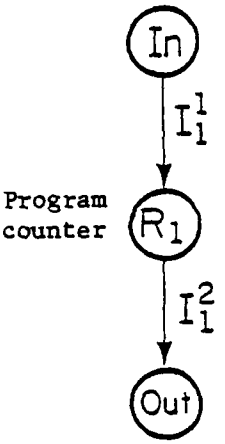
from IN to R_3 labeled I_5^4 and the self loop around R_3). Finally the result of the addition (ready in R_3) is stored in the main memory location pointed to by register R_1 (accounting for the other edge from R_1 to OUT labeled I_5^5 and the R_3-OUT edge). □

We now illustrate the representation of instructions of class B.

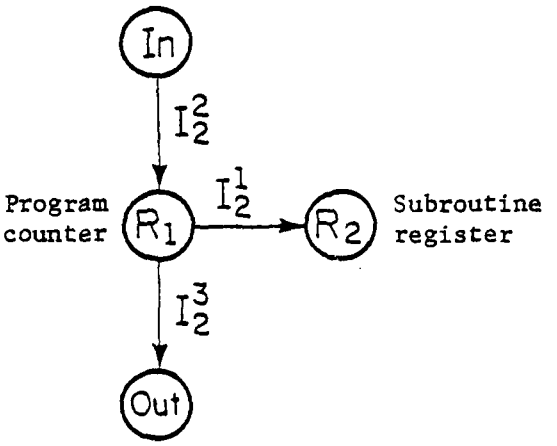
Example 2.4: Figure 2.6(a) represents a "Jump" instruction I_1. The edge from the node IN to R_1 (representing the program counter) represents the transfer of the jump address from the main memory to the program counter. The R_1-OUT edge represents the transfer of the jump address from the program counter to the address register of the main memory achieving the jump in the program sequencing.

Strictly speaking, the R_1-OUT edge (indicating the jump in the program sequencing) represents the transfer of the address of a main memory location for fetching a new instruction. Therefore the R_1-OUT edge really represents a data flow involved during the fetching of the new instruction. We take the flexible viewpoint that the data flow denoted by R_1-OUT edge could also be considered involved during the execution of the instructions of class B affecting the regular program sequencing. This viewpoint also provides a directed edge from the node representing the program counter to the OUT node, avoiding the awkward situation in which the OUT node would not be reachable from the node representing the program counter.

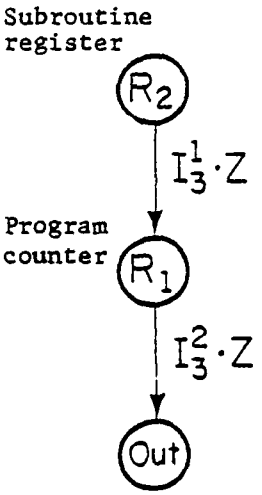
Figure 2.6(b) represents a "Jump to subroutine" instruction I_2 where the return address for the subroutine is stored in a local register called the subroutine register (denoted by R_2). The program counter is represented by node R_1. The IN-R_1 edge represents the transfer of the jump address from the main memory to the program counter, and the R_1-R_2



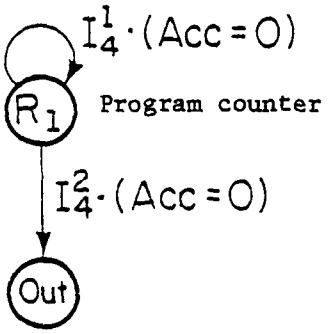
(a). I_1- Jump instruction



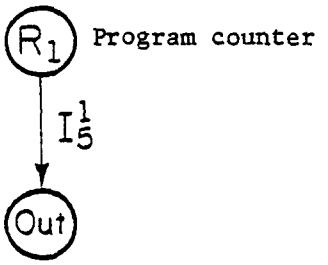
(b). I_2- Jump to subroutine instruction



(c). I_3- Return from subroutine if bit Z is set instruction



(d). I_4- Skip if the accumulator = zero instruction



(e). I_5- No operation instruction

FP-6458

Figure 2.6. Representation of instructions of class B.

edge represents the transfer of the contents of the program counter to the subroutine register, i.e., saving the return address. The return address must be saved before the jump address is transferred from the main memory to the program counter. The R_1-OUT edge represents the jump in the program sequencing. The labels of edges I_2^1, I_2^2, I_2^3 indicate the precedence relations in the data flow.

Instructions causing only conditional changes in the program sequencing can be suitably represented by tagging instruction labels with the appropriate condition code (predicate). For example, Figure 2.6(c) represents a "Return from subroutine if bit Z is set" instruction I_3. If Z = 1, the contents of the subroutine register R_2, are transferred to the program counter R_1. The R_1-OUT edge represents the jump (conditional) in the program flow.

Figure 2.6(d) shows a "Skip if the accumulator equals zero" instruction I_4. The predicate can be denoted by "ACC = 0." The node R_1 represents the program counter. The self loop around node R_1 denotes the conditional skip, i.e., the program counter is incremented if the accumulator equals zero. Figure 2.6(e) shows a "No operation" instruction I_5. The R_1-OUT edge represents the transfer of the contents of the program counter to the address register of the main memory to fetch the next instruction in the regular program sequencing.

Those instructions of class B which do not change the processor status word but only change the logic level on some status pins such as "Interrupt enable" instruction are not represented in the S-graph. □

2.3. An Example Microprocessor

We now illustrate the generation of the S-graph for a small hypothetical microprocessor. This example will also be used in Chapter 3 to demonstrate the effects of faults on the S-graph, and in Chapter 4 to illustrate various test generation procedures. Figure 2.7 shows the block diagram of this microprocessor. It has an accumulator (R_1), a general purpose register (R_2), a scratch-pad register (R_3), a program counter (R_6) and a subroutine register (R_7) to save the return address of subroutines, allowing a single level of subroutine nesting. A stack pointer (R_4) is provided which points to the top of a LIFO stack maintained in main memory. An address buffer register (R_5) is provided to store the address of operands. The ALU is capable of performing ADD, logical AND, SHIFT and COMPLEMENT operations. The instruction repertoire contains 21 instructions which are listed in Table 2.1. Though all the architectural features of the example microprocessor may not be present in any real microprocessor, they have been carefully chosen to illustrate some of the subtle points involved in test generation. It may be noticed that instructions I_1, I_2, I_3, I_5, I_6, I_7, I_8, I_{15}, I_{16}, I_{17}, I_{18} and I_{19} belong to class T, instructions I_4, I_{11}, I_{12} and I_{13} constitute class M, while class B contains instructions I_9, I_{10}, I_{14}, I_{20} and I_{21}.

Example 2.5: The S-graph for the microprocessor shown in Figure 2.7 is drawn in Figure 2.8. The self loops around R_4, labeled I_{16}^2 and I_{18}^1, represent the stack pointer incrementing and decrementing functions, respectively. The self loop around R_6, labeled $I_{10}^1 \cdot (R_1 = 0)$, represents the program counter incrementing function during the "Skip" instruction I_{10}, if the condition "register $R_1 = 0$" is satisfied. All the other edges of Figure 2.8 are self-explanatory. □

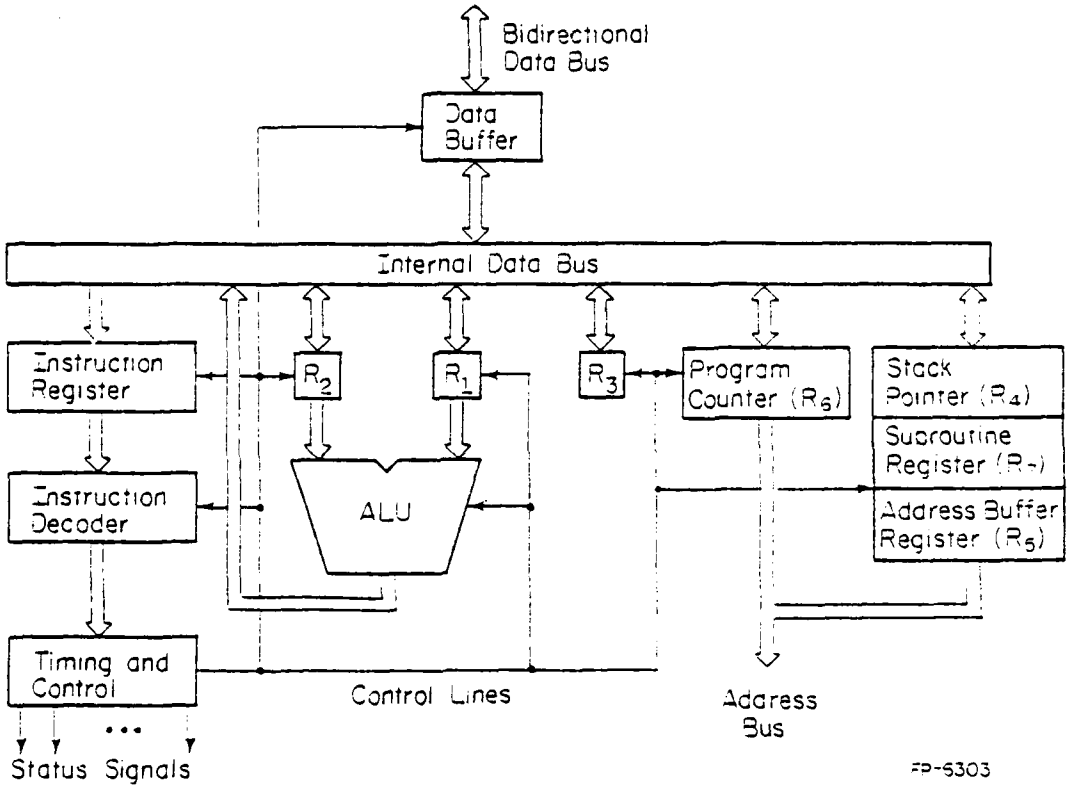


Figure 2.7. The block diagram of an example microprocessor.

Table 2.1. The instruction repertoire of the example microprocessor.

I_1 - Load register R_1 from the main memory using immediate addressing.

I_2 - Load register R_2 from the main memory using immediate addressing.

I_3 - Transfer the contents of register R_1 to register R_2.

I_4 - Add the contents of registers R_1 and R_2 and store the results in register R_1.

I_5 - Transfer the contents of register R_1 to register R_3.

I_6 - Transfer the contents of register R_3 to register R_1.

I_7 - Store register R_1 into the main memory using implied addressing.

I_8 - Store register R_2 into the main memory using implied addressing.

I_9 - Jump instruction.

I_{10} - Skip if the contents of register R_1 are zero.

I_{11} - Left shift register R_1 by one bit.

I_{12} - Complement (bit-wise) the contents of register R_1.

I_{13} - Logical AND the contents of registers R_1 and R_2 and store the result in register R_1.

I_{14} - No operation instruction.

I_{15} - Load the stack pointer (P_4) from the main memory using immediate addressing.

I_{16} - PUSH register R_1 on the LIFO stack maintained in the main memory.

I_{17} - Store register R_2 into the main memory using direct addressing.

I_{18} - POP the top of the LIFO stack and store it in R_1.

I_{19} - Load register R_2 from the main memory using direct addressing.

I_{20} - Jump to subroutine (return address is saved in the subroutine register R_7).

I_{21} - Return from subroutine.

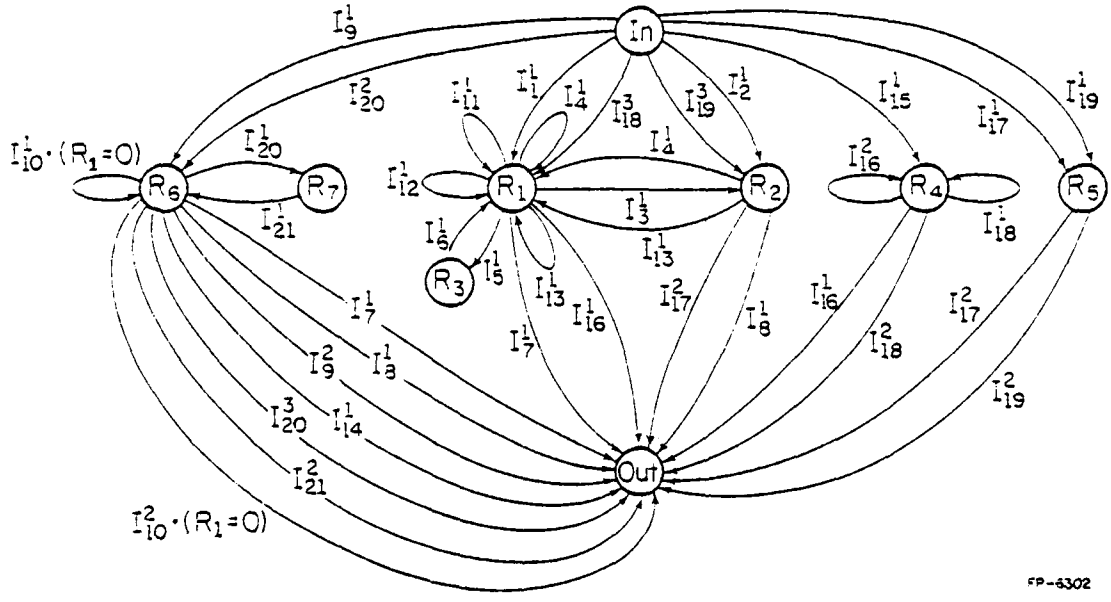


Figure 2.8. S-graph of the example microprocessor.

2.4. Definitions and Notation

Some registers of the microprocessor can be written (loaded with required data) or read out (i.e., its contents can be stored in the main memory or sent to an I/O device) by executing an explicit instruction. Examples of such registers are accumulator and general-purpose registers. On the other hand, some registers cannot be written or read out by executing any explicit instruction. For example, the address buffer register of Figure 2.5(c) can be written as well as "read out" (on the address bus) only implicitly during the execution of instruction I_3. Similarly, the stack pointer (R_2) of Figure 2.5(b) can be read out implicitly on the address bus during the execution of instruction I_2. Note that in Figure 2.8 also, the stack pointer (R_4) can be read out only implicitly during the execution of instructions I_{16} or I_{18}, though it is possible to write it explicitly by executing instruction I_{15}. The data buffer register (R_3) of Figure 2.5(e) can be written or read out only implicitly. The subroutine register (R_7) in Figure 2.8 can be read out only implicitly by executing the "Return from subroutine" instruction I_{21}. Finally, the program counter can be written only implicitly during the execution of an instruction of class B which alters the normal program sequencing.

We assume that any register can be written (implicitly or explicitly) as well as read out (implicitly or explicitly) using a sequence of instructions of class T or using an instruction of class B. This assumption can be easily justified for current microprocessors [Cush77]. In terms of the S-graph, there exists a path from the IN node to every node (representing a register) consisting of edges representing instructions of

class T or class B. Similarly there exists a path from every node to the OUT node consisting of edges representing instructions of class T or class B.

Transfer mechanisms such as buses are used to transfer data between registers, functional units, main memory, and I/O devices during the execution of an instruction. Since a test designer working in a user environment may not know the details of implementation of the transfer mechanisms, or how they are shared or time-multiplexed among different data transfers, we "map" a physical transfer mechanism used during the execution of an instruction onto a set of logical entities called transfer paths. We illustrate how to perform this mapping by means of Example 2.6 below. The set of transfer paths associated with instruction I_j is denoted by $T(I_j)$. The motivation for presenting the notion of transfer paths is to be able to develop a fault model for the data transfer function independent of the actual implementation details of the transfer mechanisms.

Example 2.6: With reference to instruction I_4 in Figure 2.8, $T(I_4)$ contains three transfer paths, two paths for transferring the contents of R_1 and R_2 to the ALU and one path for transferring the output of the ALU to R_1. $T(I_6)$ contains only one transfer path for transferring the contents of R_3 to R_1, while $T(I_{19})$ contains three transfer paths, one for transferring data (which is actually the address of an operand) from the main memory to R_5, one for transferring the address from R_5 to the address register of the main memory, and the third one to transfer data from the main memory to R_2. $T(I_{21})$ contains two transfer paths, one for transferring the contents of the subroutine register (R_7) to the program

counter, and the second one for transferring the contents of the program counter to the address register of the main memory. □

The set of source registers for an instruction I_j is defined to be that set of registers which provide the operands for instruction I_j during its execution. This set is denoted by $S(I_j)$. Similarly, the set of destination registers for an instruction I_j is defined to be that set of registers which are changed by instruction I_j during its execution. This set is denoted by $D(I_j)$. Extending this notation further, $S(I_1, I_2, \ldots, I_n) = S(I_1) \cup S(I_2) \cup \ldots \cup S(I_n)$. $D(I_1, I_2, \ldots, I_n)$ can be defined analogously. $|S(I_j)|$ and $|D(I_j)|$ denote the cardinality of the corresponding sets.

Example 2.7: In the S-graph of Figure 2.8, $S(I_7) = \{R_1\}$, $S(I_4) = \{R_1, R_2\}$, $S(I_6) = \{R_3\}$, $S(I_2) = \{IN\}$, $S(I_{21}) = \{R_7\}$, etc. Similarly $D(I_7) = \{OUT\}$, $D(I_4) = D(I_{11}) = D(I_{12}) = \{R_1\}$, $D(I_{16}) = \{R_4, OUT\}$, $D(I_{17}) = \{R_5, OUT\}$, $D(I_{20}) = \{R_6, R_7, OUT\}$. $|S(I_4)| = 2$, $|D(I_{20})| = 3$. □

The set of directed edges denoting an instruction I_j in the S-graph is called its edge set and is denoted by $E(I_j)$. READ (R_i) denotes the shortest sequence of instructions of class T or class B that is necessary to read out register R_i (implicitly or explicitly). Similarly WRITE (R_i) denotes the shortest sequence of instructions of class T or class B that is necessary to write register R_i (implicitly or explicitly). $|E(I_j)|$, $|\text{READ } (R_i)|$, and $|\text{WRITE } (R_i)|$ denote the cardinality of the corresponding set or sequences.

Example 2.8: For the S-graph shown in Figure 2.8, READ $(R_1) = \langle I_7 \rangle$, READ $(R_3) = \langle I_6, I_7 \rangle$, READ $(R_5) = \langle I_{17} \rangle$, READ $(R_7) = \langle I_{21} \rangle$, etc. WRITE $(R_1) = \langle I_1 \rangle$, WRITE $(R_5) = \langle I_{17} \rangle$, WRITE $(R_3) = \langle I_1, I_5 \rangle$, WRITE $(R_7) = \langle I_9, I_{20} \rangle$. Thus, $|\text{READ } (R_3)| = 2$, $|\text{WRITE } (R_7)| = 2$. □

We allow $|D(I_j)| > 1$ only if instruction I_j involves data transfer between the main memory (or an I/O device) and registers of the microprocessor during its execution. Thus $|D(I_j)| = 1$ for all those instructions which do not involve data transfer between the main memory and some registers during their execution. We need not consider $|D(I_j)| > 1$ in the case of these instructions, because the results of instructions of class M or T are usually not stored in more than one register. This does not mean that $|D(I_j)| > 1$ for every instruction I_j which causes data transfer to take place between the main memory and registers during its execution.

Thus we have constructed a model based on the data flow involved during the execution of an instruction satisfying the first requirement given in Section 2.2. The S-graph depends on the instruction repertoire and the functions performed by it, i.e., the S-graph reflects the architecture of the microprocessor. As will be described in Chapter 4, this feature makes it possible to consider the instruction set and organization as parameters of the test generation procedures. This satisfies the second requirement given in Section 2.2. The third requirement is related to the development of a fault model defined at functional level. This is the topic of Chapter 3.

2.5. Study of Data Transfers Among Registers

In this section we develop a framework to study how the contents of registers in the microprocessor change when a sequence of instructions of class T (called by the generic name T sequence) is executed. Specific occurrences of the T sequence are denoted by symbols σ, σ_1, σ_2, etc. A T sequence σ is specified by listing its component instructions, i.e., $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$; I_{j_1} is executed first, followed by I_{j_2}, and so on.

We denote this by $I_{j_1} < I_{j_2} < \ldots < I_{j_n}$. An instruction may occur more than once in a T sequence. Since we are considering data transfers among registers only, we concentrate only on those instructions of class T which transfer data among registers, and not between the main memory and registers. Results derived in this section will be used in Section 4.3.3 for generating tests to detect faults in the instruction decoding and control function, and for proving their fault coverage.

Definition 2.1: Register R_i is 1-step transferrable to register R_j under a T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$, if the contents of R_i before the execution of the sequence become the final contents of R_j at the end of the execution of the sequence. Such a register R_i is denoted as $R_j^1(\sigma)$. □

Lemma 2.1: Given a T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$ and register R_j, there exists one and only one register $R_j^1(\sigma)$.

Proof: Follows immediately from Definition 2.1. □

Example 2.9: Consider a hypothetical S-graph shown in Figure 2.9. Consider the T sequence $\sigma = \langle I_0, I_1, I_2, I_3, I_4, I_5 \rangle$. We have $R_6^1(\sigma) = R_0$, and $R_0^1(\sigma) = R_3$. □

If the T sequence σ, in the example above, is executed one more time the contents of R_3 (before the first execution of the T sequence) would become the final contents of R_6. This observation motivates the next definition.

Definition 2.2: Register R_i is K-step transferrable to register R_j under a T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$, if the contents of R_i before the first execution of the sequence become the final contents of R_j at the end of the K^{th} execution of the sequence, where K is the

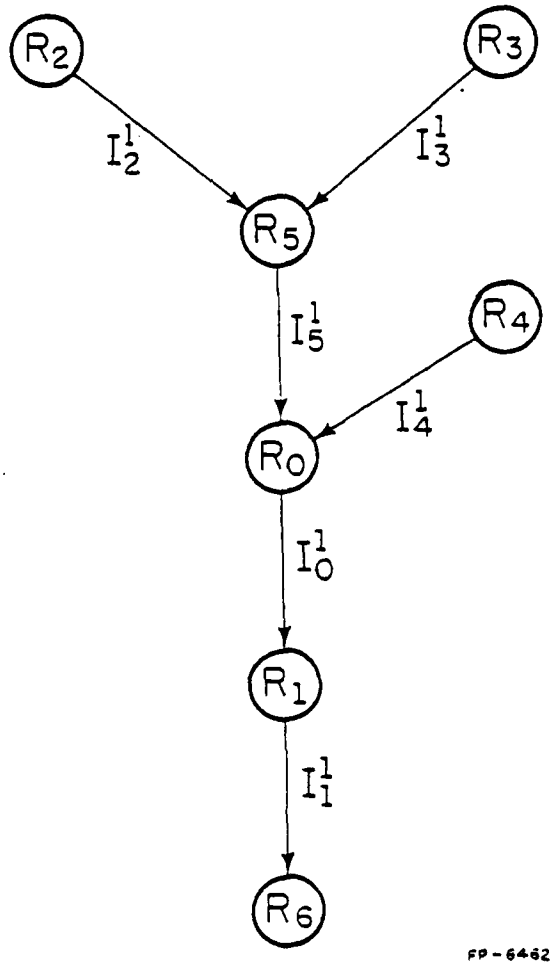


Figure 2.9. An example illustrating Definitions 2.1 and 2.2.

smallest such integer. Such a register R_i is denoted as $R_j^K(\sigma)$. □

Example 2.10: For the S-graph shown in Figure 2.9, $R_6^2(\sigma) = R_3$ under the T sequence $\sigma = \langle I_0, I_1, I_2, I_3, I_4, I_5 \rangle$. □

If we denote the T-sequence formed by concatenating two T sequences σ_1 and σ_2 as $\sigma_1 \cdot \sigma_2$, then in the context of Example 2.10, $R_6^1(\sigma \cdot \sigma) = R_6^2(\sigma) = R_3$. Therefore Definition 2.2 may appear rather contrieved and artificial because $R_j^K(\sigma) = R_j^1(\sigma \cdot \sigma \cdot \ldots \sigma)$, where the T sequence $\sigma \cdot \sigma \ldots \sigma$ is formed by concatenating K copies of T sequence σ. However, as mentioned earlier, results derived in this section will be used in Section 4.3.3 where test generation procedures are given. Some of these test generation procedures involve loops containing a T sequence. When the loop is to be executed K times, it is easier and more natural to consider the T sequence being executed K times rather than a long sequence formed by concatenating K copies of the T sequence being executed once.

M denotes the smallest integer such that if any register is K-step transferrable to a given register R_j under the T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$, then $K \leq M$.

Lemma 2.2: Given a T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$ and a register R_j, there exists one and only one register $R_j^K(\sigma)$, where $K \leq M$.

Proof: Follows from Definition 2.2. □

We are interested in finding the relation between M and *n - the number of instructions in* $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$. (Recall that $I_{j_1} < I_{j_2} < \ldots < I_{j_n}$.) In order to do this we first show how to

find $R_j^1(\sigma)$ under a given T sequence σ. Consider a set of instructions $A = \{I_{j_k} | I_{j_k} \in \sigma \text{ and } D(I_{j_k}) = \{R_j\}\}$. If set A is found to be empty, then, of course, $R_j^1(\sigma) = R_j$ because the execution of the T sequence σ does not change the contents of R_j. Moreover, M = 1. On the other hand, if set A is found to be nonempty, then choose $I_{j_p} \in A$ such that there exists no other $I_{j_q} \in A$ with $I_{j_p} < I_{j_q}$, i.e., choose I_{j_p} which is executed latest in σ but which still belongs to set A. Designate the instruction so chosen as I_{ℓ_1}. Note that I_{ℓ_1} is a unique instruction, and it is the last instruction in σ which changes the contents of R_j.

Consider a set of instructions $B_1 = \{I_{j_k} | I_{j_k} \in \sigma, D(I_{j_k}) = S(I_{\ell_1}),$ and $I_{j_k} < I_{\ell_1}\}$. If set B_1 is found to be empty, then $R_j^1(\sigma) = S(I_{\ell_1})$ because when I_{ℓ_1} is executed the contents of $S(I_{\ell_1})$ are the same as they were before the first instruction in σ was executed. (Note that I_{ℓ_1} transfers the contents of $S(I_{\ell_1})$ to R_j, and no instruction that occurs after I_{ℓ_1} in σ can change the contents of R_j.) On the other hand, if set B_1 is found to be nonempty, then choose $I_{j_p} \in B_1$ such that there exists no other $I_{j_q} \in B_1$ with $I_{j_p} < I_{j_q}$, i.e., choose I_{j_p} which is executed latest in σ but which still belongs to set B_1. Designate the instruction so chosen as I_{ℓ_2}. Note that I_{ℓ_2} is a unique instruction, and $I_{\ell_2} < I_{\ell_1}$.

Now consider a set of instructions $B_2 = \{I_{j_k} | I_{j_k} \in \sigma, D(I_{j_k}) = S(I_{\ell_2}),$ and $I_{j_k} < I_{\ell_2}\}$. If set B_2 is found to be empty, then $R_j^1(\sigma) = S(I_{\ell_2})$. This is explained as follows: When I_{ℓ_2} is executed the contents of $S(I_{\ell_2})$ are the same as they were before the first instruction in σ was executed. I_{ℓ_2} transfers the contents of $S(I_{\ell_2})$ to $S(I_{\ell_1})$ and the contents of $S(I_{\ell_1})$ do not change between the executions of I_{ℓ_2} and I_{ℓ_1}. I_{ℓ_1} transfers the contents of $S(I_{\ell_1})$ to R_j, and no instruction that occurs after I_{ℓ_1} in

σ can change the contents of R_j. Thus the contents of $S(I_{\ell_2})$ before the execution of σ become the final contents of R_j at the end of the execution of σ. On the other hand, if set B_2 is found to be nonempty, then choose $I_{j_p} \in B_2$ such that there exists no other $I_{j_q} \in B_2$, with $I_{j_p} < I_{j_q}$. Designate the instruction so choosen as I_{ℓ_3}. Note that I_{ℓ_3} is a unique instruction, and $I_{\ell_3} < I_{\ell_2}$.

This process of selecting sets A, B_1, B_2, ..., and instructions I_{ℓ_1}, I_{ℓ_2}, I_{ℓ_3}, ... can be continued until set B_i is found to be empty. Since $I_{\ell_i} < I_{\ell_{i-1}} < \ldots < I_{\ell_2} < I_{\ell_1}$, and there are n instruction in σ, the process must terminate in at most n steps, i.e., when set $B_i = \{I_{j_k} \mid I_{j_k} \in \sigma$, $D(I_{j_k}) = S(I_{\ell_i})$, and $I_{j_k} < I_{\ell_i}\}$ is found empty, the process terminates and we get $R_j^1(\sigma) = S(I_{\ell_i})$. Of course, $S(I_{\ell_i})$ could be the same as R_j, in which case M = 1.

We call the sequence of instructions $\langle I_{\ell_i}, I_{\ell_{i-1}}, \ldots, I_{\ell_1} \rangle$ the <u>characteristic sequence</u> associated with the transfer of the contents of $R_j^1(\sigma)$ to R_j, and denote it by σ_1. Since each instruction in σ_1 is a unique instruction, the characteristic sequence σ_1 is also a unique sequence. Note that σ_1 is a subsequence of σ. Let the initial contents of $R_j^1(\sigma)$ (i.e., the contents before the first instruction in σ is executed) be denoted by d_1. During the execution of σ, instructions in $\sigma_1 = \langle I_{\ell_i}, I_{\ell_{i-1}}, \ldots, I_{\ell_1} \rangle$ form a "chain" of instructions transferring the initial contents of $R_j^1(\sigma)$ to R_j, i.e., I_{ℓ_i} transfers data d_1 from $R_j^1(\sigma)$ to $S(I_{\ell_{i-1}})$, $I_{\ell_{i-1}}$ transfers data d_1 from $S(I_{\ell_{i-1}})$ to $S(I_{\ell_{i-2}})$, ... , finally I_{ℓ_1} transfers data d_1 from $S(I_{\ell_1})$ to R_j. Concisely we may say that during the execution of σ, each instruction in σ_1 transfers data d_1 where d_1 represents the initial contents of $R_j^1(\sigma)$.

Example 2.11: Consider a hypothetical S-graph shown in Figure 2.10. Consider the T sequence $\sigma = \langle I_8, I_{14}, I_1, I_{11}, I_2, I_6, I_9, I_3, I_{13}, I_4, I_{13}, I_{10}, I_{12}, I_5 \rangle$. Note that instruction I_{13} is executed twice in the sequence. Of course, in the S-graph it is represented only once using the edge from node R_4 to node R_3. In this example

$A = \{I_{11}, I_{12}\}$; $I_{\ell_1} = I_{12}$; $S(I_{\ell_1}) = \{R_2\}$.

$B_1 = \{I_8, I_{10}\}$; $I_{\ell_2} = I_{10}$; $S(I_{\ell_2}) = \{R_3\}$.

$B_2 = \{I_6, I_9, I_{13}\}$; $I_{\ell_3} = I_{13}$; $S(I_{\ell_3}) = \{R_4\}$.

$B_3 = \{I_3, I_4\}$; $I_{\ell_4} = I_4$; $S(I_{\ell_4}) = \{R_3\}$.

$B_4 = \{I_6, I_9, I_{13}\}$; $I_{\ell_5} = I_{13}$; $S(I_{\ell_5}) = \{R_4\}$.

$B_5 = \{I_3\}$; $I_{\ell_6} = I_3$; $S(I_{\ell_6}) = \{R_6\}$.

$B_6 = \{I_2, I_{14}\}$; $I_{\ell_7} = I_2$; $S(I_{\ell_7}) = \{R_7\}$.

$B_7 = \{I_1\}$; $I_{\ell_8} = I_1$; $S(I_{\ell_8}) = \{R_4\}$.

$B_8 = \{\phi\}$.

Hence $R_1^1(\sigma) = S(I_{\ell_8}) = R_4$. Note that I_{ℓ_3} refers to the second occurrence of I_{13} in σ, while I_{ℓ_5} refers to its first occurrence. The characteristic sequence $\sigma_1 = \langle I_1, I_2, I_3, I_{13}, I_4, I_{13}, I_{10}, I_{12} \rangle$ which is a subsequence of σ. □

Now we show how to find $R_j^K(\sigma)$ under the T sequence σ (for $K \leq M$). If a register which is 1-step transferrable to $R_j^1(\sigma)$ under the T sequence σ does not belong to the set $\{R_j, R_j^1(\sigma)\}$, it must be

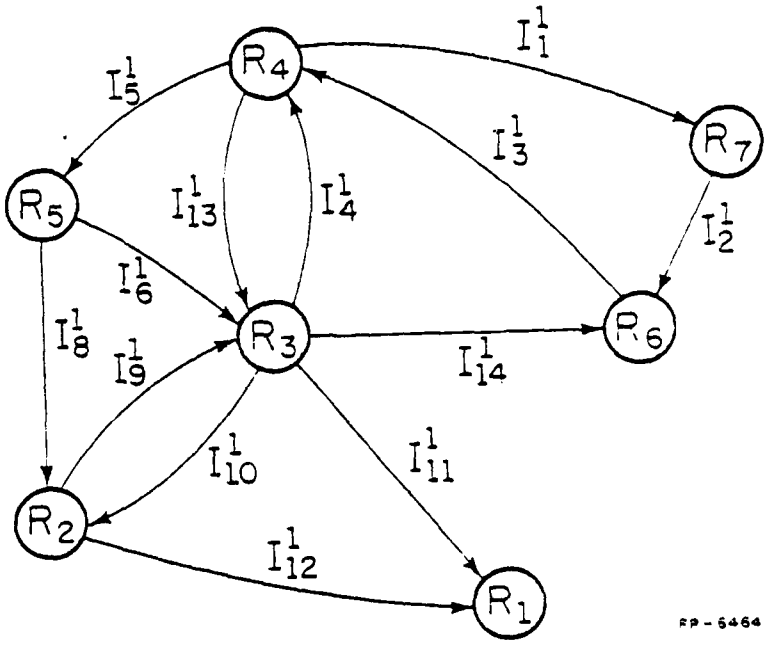


Figure 2.10. S-graph for Example 2.11.

2-step transferrable to R_j and is designated as $R_j^2(\sigma)$. On the other hand, if it belongs to the set $\{R_j, R_j^1(\sigma)\}$ it is 1-step transferrable to R_j, allowing us to conclude that M = 1. Extending this argument, if a register which is 1-step transferrable to $R_j^K(\sigma)$ under the T sequence σ does not belong to the set $\{R_j, R_j^1(\sigma), R_j^2(\sigma), \ldots, R_j^K(\sigma)\}$, it must be (K+1)-step transferrable to R_j and is designated as $R_j^{K+1}(\sigma)$. On the other hand, if it belongs to the set $\{R_j, R_j^1(\sigma), R_j^2(\sigma), R_j^2(\sigma), \ldots, R_j^K(\sigma)\}$ it is p-step transferrable to R_j, where $1 \leq p \leq K$. (Refer to Definition 2.2.) In this case we can immediately conclude that M = K.

We denote the characteristic sequence associated with the transfer of the contents of $R_j^i(\sigma)$ to $R_j^{i-1}(\sigma)$ by σ_i, for $2 \leq i \leq M$. Concisely we may say that during the execution of σ, each instruction in σ_i transfers data d_i where d_i represents the initial contents of $R_j^i(\sigma)$, for $2 \leq i \leq M$. This discussion leads to the following lemma.

Lemma 2.3: $R_j^i(\sigma)$ is 1-step transferrable to $R_j^{i-1}(\sigma)$, for $2 \leq i \leq M$. Some register in the set $\{R_j, R_j^1(\sigma), R_j^2(\sigma), \ldots, R_j^M(\sigma)\}$ is 1-step transferrable to $R_j^M(\sigma)$. □

Definition 2.3: Let $\sigma_i = \langle I_{j_a}, I_{j_b}, I_{j_c}, \ldots \rangle$ and $\sigma_j = \langle I_{j_p}, I_{j_q}, I_{j_r}, \ldots \rangle$ be two subsequence of a T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$. σ_i and σ_j are defined to be disjoint if the sets {a, b, c, ...} and {p, q, r, ...} are disjoint, i.e., {a, b, c, ...} $\cap$ {p, q, r, ...} = $\emptyset$. If {a, b, c, ...} $\cap$ {p, q, r, ...} $\neq \emptyset$, σ_i and σ_j are not disjoint. □

Definition 2.4: Let $\sigma_i, \sigma_j, \sigma_k, \ldots$ be subsequences of a T sequence σ. Subsequences in the set $\{\sigma_i, \sigma_j, \sigma_k, \ldots\}$ are defined to be mutually disjoint if each pair of subsequences is disjoint. They are

not mutually disjoint if any pair of subsequences is not disjoint. □

Lemma 2.4: A T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots, I_{j_n} \rangle$ can be partitioned into a set containing at most n mutually disjoint subsequences. □

Lemma 2.5: Subsequences in the set $\{\sigma_1, \sigma_2, \ldots, \sigma_M\}$ are mutually disjoint, where σ_1 denotes the characteristic sequence associated with the transfer of $R_j^1(\sigma)$ to R_j, and σ_i denotes the characteristic sequence associated with the transfer of $R_j^i(\sigma)$ to $R_j^{i-1}(\sigma)$, for $2 \leq i \leq M$.

Proof: During the execution of σ, each instruction in σ_i transfer data d_i where d_i represents the initial contents of $R_j^i(\sigma)$ for $1 \leq i \leq M$. Let us assume the contrary, i.e., subsequences in the set $\{\sigma_1, \sigma_2, \ldots, \sigma_M\}$ are not mutually disjoint. Therefore at least one pair of subsequences, say σ_i and σ_k, must not be disjoint. In this case, at the end of execution of σ, either the contents of $R_j^{i-1}(\sigma)$ are different from d_i, or the contents of $R_j^{k-1}(\sigma)$ are different from d_k. This contradicts the assumption that σ_i is the characteristic sequence associated with the transfer of $R_j^i(\sigma)$ to $R_j^{i-1}(\sigma)$, and σ_k is the characteristic sequence associated with the transfer of $R_j^k(\sigma)$ to $R_j^{k-1}(\sigma)$. Therefore subsequences in the set $\{\sigma_1, \sigma_2, \ldots, \sigma_M\}$ must be mutaully disjoint. □

Theorem 2.1: $M \leq n$, where n = the number of instructions in the T sequence σ.

Proof: Follows immediately from Lemmas 2.4 and 2.5. □

Corollary 2.1: Let the initial contents of registers $R_j^1(\sigma)$, $R_j^2(\sigma)$, ... $R_j^M(\sigma)$ be d_1, d_2, ..., d_M, respectively. Then at the end of K^{th} execution ($1 \leq K \leq M$) of the T sequence σ, register R_j will

contain d_K. At the end of i^{th} execution ($i > M$) of the T sequence σ, R_j will contain some data belonging to the set $\{d_1, d_2, \ldots, d_M\}$.

Proof: Follows from Lemma 2.3. □

Corollary 2.2: If register R_j contains the same data d at the end of each of i executions of the T sequence $\sigma = \langle I_{j_1}, I_{j_2}, \ldots I_{j_n} \rangle$, for $1 \leq i \leq n$, then at the end of the $(n+p)^{th}$ execution of the T sequence σ, for $p \geq 1$, R_j will contain the same data d.

Proof: Follows from Corollary 2.1, and $M \leq n$. □

Corollary 2.3: If register R_j contains the same data d at the end of each of i executions of the T sequence σ containing at most K-1 instructions, for $1 \leq i \leq K-1$, then at the end of K^{th} execution of the T sequence σ, R_j will contain the same data d.

Proof: Follows immediately from Corollary 2.2. □

3. FUNCTIONAL LEVEL FAULT MODELS FOR MICROPROCESSORS

In this chapter we present fault models for various functions in a microprocessor in accordance with the third requirement cited in Section 2.2. We develop fault models which are quite independent of the implementation details of the microprocessor. We categorize various functions in a microprocessor into the register decoding function, instruction decoding and control function, data storage function, data transfer function, and data manipulation function. We will present a fault model for each of these functions at a higher level. We will, however, point out the underlying fault mechanisms in order to clarify the reasons for choosing the particular models. We will also describe the effects of these faults at the level of the graph-theoretic model for a microprocessor presented in Chapter 2.

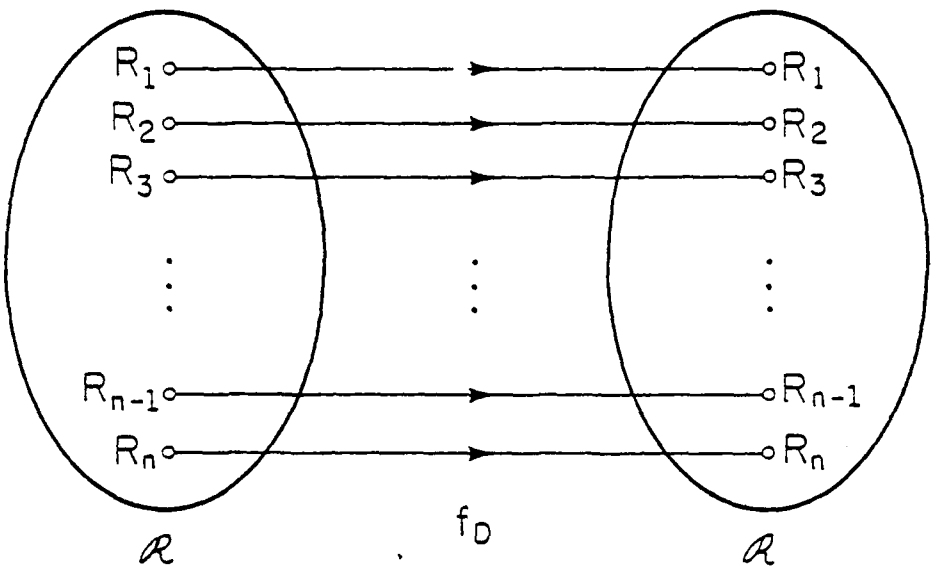
3.1. Fault Model for the Register Decoding Function

Registers on a microprocessor chip are typically realized as small random-access memories (RAM) [INTE75]. They could also be realized as separate registers interconnected by a network of multiplexers, demultiplexers and buses. Various instructions use registers to fetch operands or address of operands and to store results of operations. Register decoding refers to the task of decoding the "address" of a register which may be stored as a specific bit pattern in the instructions involving that register or which may be generated by the control unit during the execution of the instructions. We want to develop a fault model for this decoding function independent of the realization of the decoding mechanism.

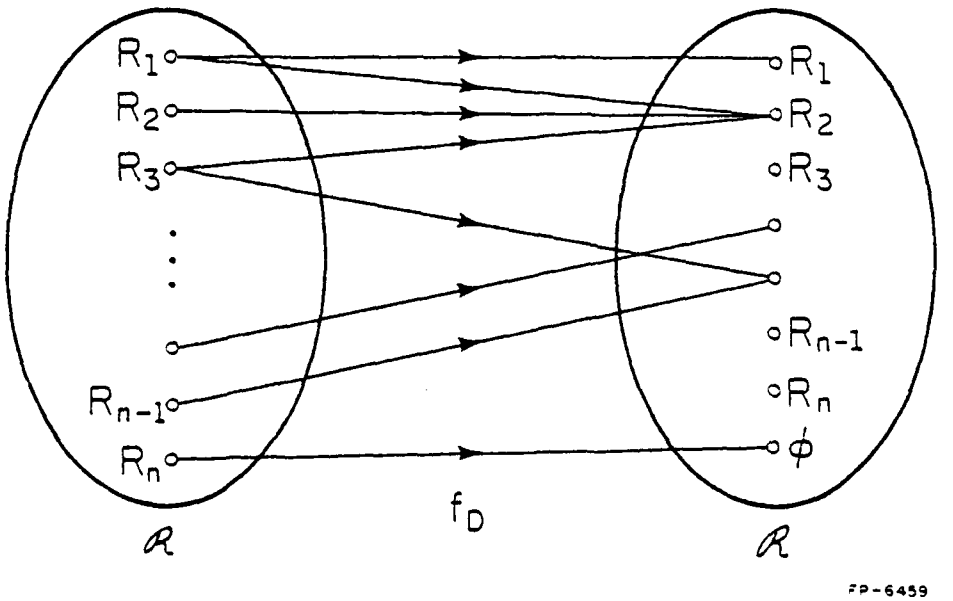
The register decoding function can be modeled as a mapping f_D from $\Re$ to $\Re \cup \{\emptyset\}$, where $\emptyset$ denotes a null or nonexistent register. Let $f_D(R_i) \subseteq \Re \cup \{\emptyset\}$ denote the set of registers which is the image of R_i under the mapping f_D. If there is no fault in the register decoding function we get $f_D(R_i) = \{R_i\}$, for every $R_i \in \Re$. Under a fault, if $f_D(R_i) = \{\emptyset\}$, whenever register R_i is to be accessed (while executing any instruction which involves R_i), no register is accessed. Obviously if $\emptyset \in f_D(R_i)$ then $f_D(R_i) = \{\emptyset\}$ because $f_D(R_i) = \{R_j, R_k, \ldots, \emptyset\}$ is meaningless. If $f_D(R_i) \neq \{\emptyset\}$ then whenever R_i is accessed, all the registers in the set $f_D(R_i)$ are accessed. By this we mean, whenever R_i is to be written with data d, all the registers in $f_D(R_i)$ would be written with data d, and whenever the contents of R_i are to be retrieved or used, the contents formed by the bit-wise OR or AND function (depending on technology) over the registers of the set $f_D(R_i)$ will be retrieved. Under this fault we allow $f_D(R_i) \neq \{R_i\}$, for every $R_i \in \Re$.

This situation can be best illustrated by means of a pictorial representation shown in Figure 3.1. In Figure 3.1(a) the mapping f_D is shown under the condition that there is no fault in the register decoding function. Under this condition, f_D is a one-to-one correspondence from $\Re$ to $\Re$. When there is a fault in the register decoding function f_D becomes, in general, a many-to-many correspondence from $\Re$ to $\Re \cup \{\emptyset\}$. This is illustrated in Figure 3.1(b).

We now briefly comment on the fault mechanisms responsible for faults in the register decoding function. Consider a subset of registers $\Re' \subseteq \Re$ which is realized as a random-access memory on the microprocessor chip. Due to faults in the address decoder in this memory some registers



(a). f_D is a one-to-one correspondence from $\mathcal{R}$ to $\mathcal{R}$ if there is no fault in the register decoding function.



FP-6459

(b). f_D is a many-to-many correspondence from $\mathcal{R}$ to $\mathcal{R} \cup \{\phi\}$ if there are faults in the register decoding function.

Figure 3.1. Representation of the mapping f_D.

in $\Re'$ could be decoded as some other registers in $\Re'$ [NTAb78]. This can be adequately modeled by a many-to-many correspondence from $\Re'$ to $\Re' \cup \{\phi\}$. Of course, under this fault a register in $\Re'$ cannot be decoded as some other register not in $\Re'$.

In order to rule out the possibility of a register R_1 being decoded as another register R_2 under a fault, we must know whether R_1 and R_2 belong to different random-access memories realized on the chip, or we must know of the existence of some mechanism (realizing R_1 and R_2) under which this fault cannot be present, i.e., we need to know the implementation details. Our desire is to make the fault model as independent as possible of the actual implementation. Therefore we allow f_D to be a many-to-many correspondence from $\Re$ to $\Re \cup \{\phi\}$ under a fault. We are thus considering the "worst case" behavior under the register decoding faults.

Alternatively registers could be realized as separate registers and interconnected with a network of multiplexers, demultiplexers and buses. A typical situation is shown in Figure 3.2, where it is desired to choose one register from a subset of registers $\Re_1 = \{R_1, R_2, \ldots, R_k\} \subset \Re$ for transferring its contents to a register to be chosen from another subset of registers $\Re_2 = \{R_{k+1}, R_{k+2}, \ldots, R_m\} \subset \Re$. The task is accomplished by using a k-to-1 multiplexer for choosing a source register from $\Re_1$, and a 1-to-(m-k) demultiplexer for selecting a destination register from $\Re_2$. The multiplexer and the demultiplexer receive the addresses of the registers to be selected from the control unit.

Due to faults in these units the wrong registers may be chosen, or more than one registers could be chosen. Under some fault in the control unit, an incorrect register address could be sent to the multiplexer

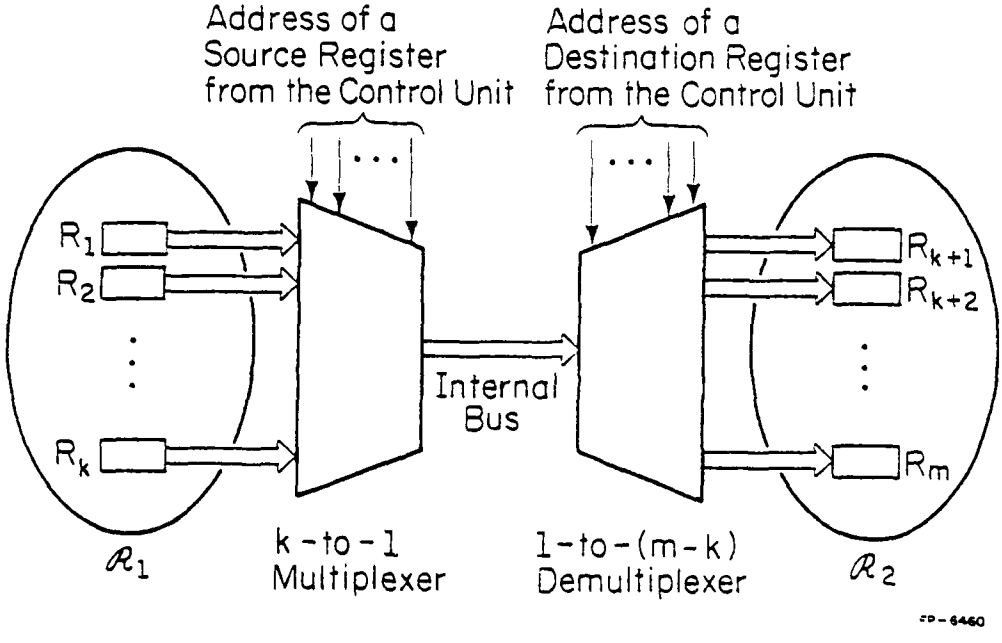


Figure 3.2. A multiplexer-demultiplexer mechanism for data transfer.

or demultiplexer resulting into the selection of a wrong register. All these faults can be adequately modeled by a many-to-many correspondence from $\mathcal{R}_1$ to $\mathcal{R}_1 \cup \{\phi\}$, and another many-to-many correspondence from $\mathcal{R}_2$ to $\mathcal{R}_2 \cup \{\phi\}$. In this particular implementation, a register in $\mathcal{R}_1$ may not be decoded as a register in $\mathcal{R}_2$, and vice-versa. We avoid all these implementation dependent details by allowing f_D to be a many-to-many correspondence from $\mathcal{R}$ to $\mathcal{R} \cup \{\phi\}$.

At this point one may wonder how a register used to store address of operands such as R_5 and R_7 (in Figure 2.8), could be decoded as a register used to store operands, such as R_1 and R_2, particularly in the light of the fact that the widths of registers used to store addresses usually differ from those used to store data. This is quite likely in the following situation.

All registers are realized as a RAM array on the microprocessor chip. Each word of the RAM is 16 bits in width and can be used as a single register for storing addresses which are 16 bits in width. A single word can also be used as a pair of registers for storing data which is 8 bits in width. This is the way registers are implemented on the INTEL 8080 microprocessor [INTE75].

We now extend the notation developed in Chapter 2. $f_D(D(I_j))$ denotes the set of registers formed by making the union of the image sets of registers in $D(I_j)$ under the mapping f_D. $f_D(S(I_j))$ can be analogously defined. Extending this notation further, $f_D(D(I_1,I_2,\ldots,I_n))$ $= f_D(D(I_1)) \cup f_D(D(I_2)) \cup \ldots \cup f_D(D(I_n))$. The set $f_D(S(I_1,I_2,\ldots,I_n))$ can be defined similarly.

We now illustrate the effects of faults in the register decoding function at the level of the S-graph by means of the following example.

Example 3.1: In terms of the S-graph of Figure 2.8, $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$. If $f_D(R_5) = \{R_1\}$, register R_1 will play the role of the address buffer register R_5 during the execution of instructions I_{17} and I_{19}. Thus under I_{17}, R_2 will be read out correctly, but the contents of register R_1 will be changed instead of that of R_5. If $f_D(R_1) = \{R_2\}$, when I_1 is executed R_2 will be written instead of R_1. Moreover, I_7 will read out R_2 instead of R_1. If $f_D(R_2) = \{R_1, R_3\}$, then I_8 will read out $R_1 * R_3$, where * denotes the bit-wise AND or OR function over registers R_1 and R_3 depending on technology. Similarly when I_2 is executed, both R_1 and R_3 will be written instead of R_2.

If $f_D(R_3) = \{\emptyset\}$, then I_5 will not change the contents of any register, and I_6 will transfer a ONE$^+$ or a ZERO$^+$ to R_1 depending on technology, instead of the contents of R_3. If $f_D(R_7) = \{\emptyset\}$, the "Jump to subroutine" instruction I_{20} will correctly execute the jump in the program sequencing, but will not save the return address into R_7. The fault will show up when the "Return from subroutine" instruction I_{21} is executed, because the program sequence will return to the main memory location whose address is ONE or ZERO depending on technology, as a ONE or a ZERO will be loaded into the program counter (R_6) instead of the contents of R_7.

$^+$ONE denotes a binary vector with each of its bits set to logic 1 and and having its width equal to that of a register, i.e., ONE = (11...1); similarly ZERO stands for (00...0).

If $f_D(R_5) = \{R_7\}$, instruction I_{20} will be executed correctly, i.e., the jump to subroutine will occur and the return address will be correctly saved in R_7, but now the correct execution of I_{21} will depend on whether I_{17} or I_{19} were executed within the subroutine. If they were executed in the subroutine, due to the fault $f_D(R_5) = \{R_7\}$, the subroutine register R_7 will be changed instead of the address buffer register R_5, resulting into the loss of the return address saved in R_7. Thus I_{21} will cause the program to branch to some location that equals the address of operand used in the last instruction I_{17} or I_{19} executed within the subroutine. □

3.2. Fault Model for the Instruction Decoding and Control Function

The instruction decoding mechanism is shown as a block diagram in Figure 3.3. Basically it is a decoder whose inputs are the instruction opcodes and whose outputs correspond to the control signals that initiate the execution of instructions. For each valid opcode, one and only one output of the decoder is activated initiating the execution of one and only one instruction.

Under a fault in the instruction decoding and control function, the faulty behavior of the microprocessor can be specified as follows. When instruction I_j is executed any one of the following can happen:

1. Instead of instruction I_j some other instruction I_k is executed. This fault is denoted by $f(I_j/I_k)$.

2. In addition to instruction I_j, some other instruction I_k is also activated. This fault is denoted by $f(I_j/I_j+I_k)$.

3. No instruction is executed. This fault is denoted by $f(I_j/\emptyset)$.

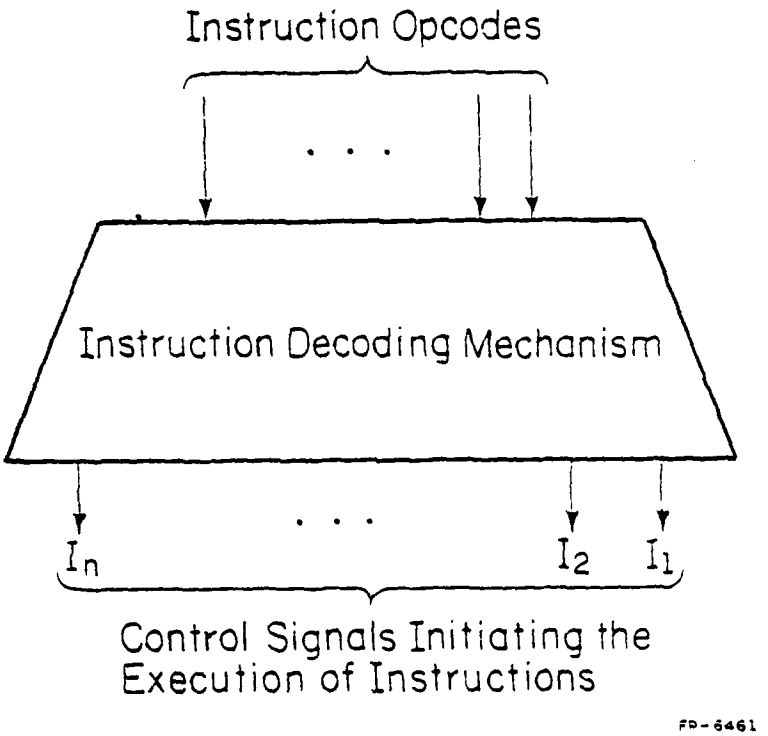


Figure 3.3. Block diagram of the instruction decoding mechanism.

This fault model is strongly motivated by the fact summarized in the following theorem.

Theorem 3.1: If a decoder is realized without any reconvergent fanout then under a single stuck-at fault its behavior can be formulated independent of its implementation details as follows: for a given valid input to the decoder, instead of, or in addition to the expected output some other output is activated, or no output is activated.

Proof: See the Appendix. □

The assumption of no reconvergent fanout in the instruction decoding mechanism is quite reasonable as it has n inputs and as many as 2^n outputs. We would like to allow the faulty behavior stated above for each instruction of the microprocessor. However, it makes the test generation procedures extremely complicated. Therefore we impose two constraints (given below as 4 and 5) on the decoder behavior under faults in the instruction decoding and control function.

4. If faults $f(I_j/I_k)$ or $f(I_j/I_j+I_k)$ are present then instruction I_k will be correctly executed.

5. If faults $f(I_j/I_k)$, $f(I_j/I_j+I_k)$ or $f(I_j/\phi)$ are present then faults $f(I_q/I_j)$ or $f(I_q/I_q+I_j)$ cannot be present.

The behavior of a decoder under a single stuck-at fault does not violate these constraints. This will also be proved in the Appendix. Any number of instructions could be faulty subject to the set of specifications 1 through 5. As an example, under the fault model, faults $f(I_1/I_2)$, $f(I_3/I_3+I_2)$, $f(I_4/I_2)$ can exist simultaneously, so can $f(I_1/I_2)$, $f(I_3/\phi)$, $f(I_4/I_6)$, $f(I_5/I_5+I_6)$. Thus, this fault model can account for all single stuck-at faults in the instruction decoding mechanism.

In practice, some faults in the instruction decoding and control function such as $f(I_j/I_k)$ or $f(I_j/I_j+I_k)$ may be readily detected if a different number of machine cycles are needed to execute instructions I_j and I_k, or different status signals are emitted during their execution [ThAb78].

We now illustrate the effects of faults in the instruction decoding function at the level of the S-graph by means of the following example.

Example 3.2: In terms of the S-graph of Figure 2.8, under fault $f(I_2/\phi)$, register R_2 will not be written, i.e., its contents remain unchanged. If $f(I_4/I_6)$ is present, then the contents of R_3 will be transferred to R_1 instead of the sum of R_1 and R_2. Under $f(I_7/I_7+I_8)$, the contents of R_1 * R_2 will be read out, where as before, * indicates the bit-wise logical OR or AND function depending on technology. If $f(I_9/I_9+I_3)$ is present, the "Jump" instruction I_9 will be executed correctly, but at the same time the contents of R_1 will also be transferred to R_2.

Note that the faults in the instruction decoding and control function cannot be treated as faults in the register decoding function. For example, $f(I_3/I_5)$ cannot be treated as R_2 being decoded as R_3 if I_8 is executed correctly. Under $f(I_{10}/I_{21})$, instead of the "Skip if the contents of R_1 are zero" instruction the "Return from subroutine" instruction is executed. Under $f(I_6/I_6+I_{13})$, (RESULT1) * (RESULT2) will be transferred to R_1, where RESULT1 and RESULT2 are the results produced by I_6 and I_{13}, respectively, and * indicates the bit-wise logical OR or AND function between RESULT1 and RESULT2.

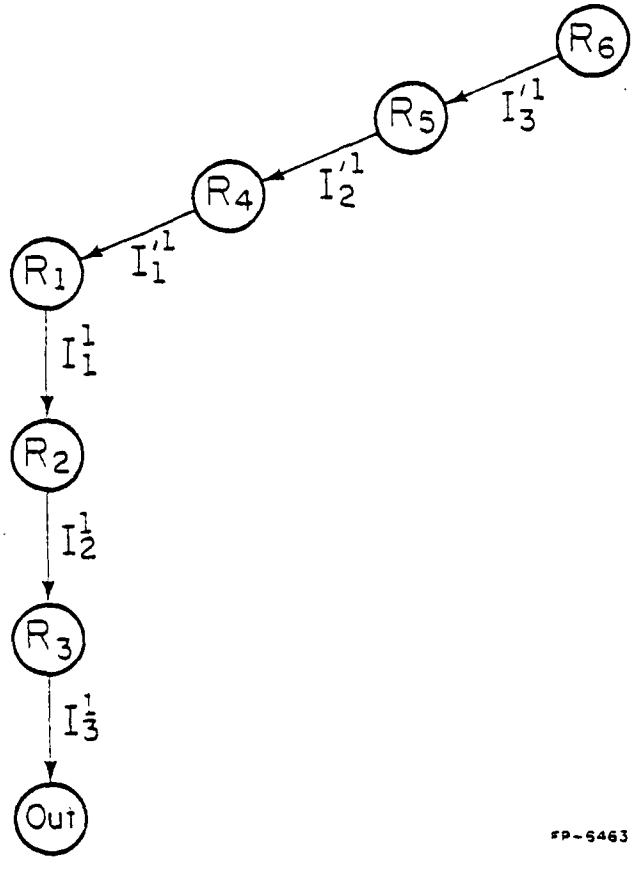
Example 3.3: This example is constructed to show how the result produced by executing a program in the presence of a multiple fault in the instruction decoding and control function differs from the one produced by executing the same program under the fault free condition. A part of the S-graph of a hypothetical microprocessor is shown in Figure 3.4(a). I_1, I_2, I_3, I_1', I_2', I_3' are instructions of class T. Note that instruction I_3 reads out register R_3. We investigate how the simultaneous existence of three faults $f(I_1/I_1+I_1')$, $f(I_2/I_2+I_2')$, and $f(I_3/I_3+I_3')$ affects the result produced by the microprocessor when it executes the program given in Figure 3.4(b). Only three instructions in the loop are shown. The loop control is given in terms of a high level language construct (FOR loop) only for conciseness and ease of understanding.

Assume that the initial contents of registers R_1, R_4, R_5, R_6 are ONE, ONE, ONE, and ZERO, respectively. Thus under the fault free condition, at the end of each of the n iterations of the loop, a ONE is read out, independent of the value of n.

Under the presence of the multiple fault described above, the program would correctly read out a ONE at the end of each of the first three iterations of the loop, but would read out a ZERO instead of a ONE at the end of each iteration after that. Therefore the program would not detect the fault if $n \leq 3$. □

3.3. Fault Model for the Data Storage Function

In this section a fault model for the data storage function is presented which accounts for the faults in various registers. We allow any cell of a register to be stuck at 0 or 1, and this fault can occur



(a). A part of the S-graph of a hypothetical microprocessor.

```
FOR   K ← 1 TO n DO
BEGIN
      .
      .
      .
      I1
      I2
      I3
      .
      .
      .
END
```

(b). The program considered in Example 3.3.

Figure 3.4. Illustrating Example 3.3.

with any number of cells of any number of registers. We now illustrate the effects of faults in the data storage function at the level of the S-graph by means of the following example.

Example 3.4: In terms of the S-graph of Figure 2.8, suppose the first and third bit of register R_1 are stuck at 1 and 0, respectively. Then, it would not be possible to store any data vector whose first and third bits are 0 and 1, respectively, in register R_1, by executing any instruction whose destination register is R_1. If the second bit of register R_7 (subroutine register) is stuck at 0, then it would not be possible to execute the "Return from subroutine" instruction I_{21} successfully if the return address has its second bit equal to 1. Thus under a fault in the data storage function some instructions may not be correctly executed for certain data and address patterns. □

3.4. Fault Model for the Data Transfer Function

In this section a fault model for the data transfer function is presented which accounts for faults in various transfer paths, i.e., buses. Under a fault in the data transfer function for any instruction I_j

1. a line in a transfer path in set $T(I_j)$ can be stuck at 0 or 1,
2. two lines of a transfer path in set $T(I_j)$ can be coupled, i.e., they fail to carry different logic values. This can happen due to metallization shorts or capacitive couplings [ThAb78].

We allow any number of transfer paths associated with any number of instructions to be faulty in this manner. This fault model is very general and is also independent of implementation details of transfer paths. Even though physical transfer mechanisms may be shared between

transfer paths in practice, by allowing each transfer path to be faulty, we are making the fault model independent of implementation details.

We now illustrate the effects of faults in the data transfer function at the level of the S-graph by means of the following example.

Example 3.5: Suppose the transfer path carrying data from the node R_2 to the OUT node (i.e., the main memory) in instruction I_8 has its second line stuck at 0, then instruction I_8 cannot be executed successfully if the data pattern stored in register R_2 has its second bit equal to 1. Suppose the transfer path used to carry the result from the ALU to register R_1 in instruction I_4 have its first and second line coupled such that the resulting logic value present on these two lines really is a logical OR or AND function (depending on technology) of the logic values that would have been present on these lines, were there no coupling.

Under this fault any ALU result whose first and second bits differ in the logic values will not be successfully transferred to R_1; if the coupling results in a logical OR function, the first and second lines of the transfer path will both carry a logic 1 when they are supposed to carry a 1 and a 0, or a 0 and a 1. Similarly if the coupling results in a logical AND function, the first and second lines of the transfer path will both carry a logic 0 when they are supposed to carry a 1 and a 0, or a 0 and a 1. Similar faults could be present with the transfer paths used to carry addresses. Thus under a fault in the data transfer function some instructions cannot be correctly executed for certain data and address patterns.

3.5. Fault Model for the Data Manipulation Function

The data manipulation function refers to various functional units such as the ALU, interrupt handling hardware, hardware used for incrementing (or decrementing) the program counter, stack pointer or index register, hardware used for computing the address of operands in various addressing modes such as indexed and relative, etc.

A microprocessor is not a network of arbitrary interconnections of these functional units. Therefore we need not really worry about the problems involved in testing a digital system comprised of a network of such functional units as mentioned in Section 2.1.3. In fact, recalling the discussion of Section 2.4 any register of a microprocessor can be read or written (explictily or implicitly) using a sequence of instructions of class T, or using an instruction of class B, i.e., the operands required for an instruction of class M can be stored in the necessary registers (or are available in the main memory) and the result produced by it can be read out from the register where it is stored by using instructions which do not belong to class M.

We do not propose any specific fault model, per se, for the data manipulation function because of the wide variety in existing designs for the ALU and other functional units such as increment or shift logic. We will assume that complete test sets can be derived for the functional units for any given fault model. The operands necessary to execute tests for a given functional unit can be stored in the proper registers by executing instructions of class T or B only, and they do not require the use of any other functional unit. Similarly the results of these tests can be read out by using instructions of class T or B only.

We allow any number of functional units to be faulty at one time.

3.6. Fault Model for Microprocessors

We now propose the fault model for microprocessors based on the fault models proposed in Sections 3.1 through 3.5. At any given time we allow the presence of *any number of faults* but only in one function described above (in Sections 3.1 through 3.5). Note that we are allowing a very general model for microprocessors (as described in Chapter 2). In addition, if we allow multiple faults in different functions, the problem becomes extremely complex. In [ThAb78], a restricted model for microprocessor was considered, (refer to Section 2.5) allowing multiple faults in different functions. In that case the problem turned out to be very complex but of manageable proportions.

4. TEST GENERATION PROCEDURES

In this chapter we present test generation procedures to generate tests for detecting faults covered by the fault models presented in Sections 3.1 through 3.5. The first step in developing test generation procedures is to assign labels to the nodes and edges of the S-graph under consideration by using the labeling algorithm given in Section 4.1. Test generation procedures for detecting faults in the register decoding function, instruction decoding and control function, data transfer and storage function, and data manipulation function are given in Sections 4.2, 4.3, 4.4 and 4.5, respectively. The fault coverage of the tests is also proved.

4.1. Algorithm 4.1: The Labeling Algorithm

This algorithm assigns integer labels to nodes and edges. The label assigned to a node representing register R_i is denoted by $\ell(R_i)$, and the label assigned to the edge set $E(I_j)$ representing instruction I_j is denoted by $\ell(I_j)$.

Step 1: Assign a label 0 to the OUT node.

Step 2: $K \leftarrow 0$;

WHILE a node remains unlabeled DO

BEGIN

Assign a label K+1 to all unlabeled nodes representing registers whose contents can be transferred (explicitly or implicitly) to any register whose node is labeled K by executing a single instruction of class T or B;

$K \leftarrow K + 1$

END

Step 3: Assign a label 1 to each edge in the set $E(I_j)$ where I_j is an instruction that reads out a register (explicitly or implicitly) during its execution.

Step 4: If I_j is an instruction whose edge set has not been labeled in step 3 then assign a label (K+1) to each edge in the set $E(I_j)$, where $\ell(D(I_j)) = K$. □

Thus the labeling algorithm first assigns an integer label to each node of the S-graph. This label (which is equal to $|\text{READ}(R_i)|$ as will be shown in Lemma 4.1) indicates the shortest "distance" of that node to the OUT node, i.e., the minimum number of instructions of class T or B that need to be executed to read out (explicitly or implicitly) the contents of the register being represented by that node. After assigning labels to the nodes of the S-graph, the labeling algorithm assigns labels to the edges representing instructions. In step 4, each edge in the set $E(I_j)$ representing instruction I_j is assigned a label K + 1, if the destination register of I_j was assigned a label K in step 1 or 2. Note that in step 4, the edge sets of only those instructions are labeled which do not cause data transfers from registers of the microprocessor to the main memory or an I/O device during their execution. For such an instruction I_j, $|D(I_j)| = 1$. (Recall the discussion in Section 2.4.)

On the other hand, the destination set $D(I_j)$ of an instruction I_j that reads out (explicitly or implicitly) a register during its execution may contain more than one register. Since the nodes representing these registers may have different labels, step 4 cannot be applied in this

case. In this case we use step 3 to assign a label 1 to each edge in the set $E(I_j)$. The choice of the label may be explained by the fact that in this case the OUT node has to be a member of the set $D(I_j)$, and step 1 assigns a label 0 to the OUT node.

For concise description, we will use the phrase "register R_i with label K," if the node representing register R_i is labeled K. Similarly we will use the phrase "instruction I_j with label K," if the edge set $E(I_j)$ representing instruction I_j is labeled K. The phrase "Execute READ (R_i)" means execute instructions in the READ (R_i) sequence; the phrase "Execute WRITE (R_i)" can be interpreted in a similar fashion.

Lemma 4.1: a) If $\ell(R_i) = K$, $|\text{READ } (R_i)| = K$. b) If $\ell(I_j) = 1$, I_j reads out (explicitly or implicitly) a register with label 1.

Proof: a) Nodes are labeled in step 2 of the labeling algorithm. A node is labeled 1 if the register represented by it can be read out (explicitly or implicitly) by executing a single instruction of class T or B. A node is labeled 2 if the contents of the register represented by it can be transferred to a register whose node is labeled 1 by executing a single instruction of class T or B, and the former register (whose node is labeled 2) cannot be read out by executing a single instruction of class T or B. Thus a register R_i whose node is labeled 2 can be read out by executing a sequence of instructions of class T or B containing two instructions and no shorter sequence exists to read it out. Therefore $|\text{READ } (R_i)| = 2$. (Recall the definition of READ (R_i) in Section 2.4.) Extending the argument in this fashion it can be easily proved that a register R_i whose node is labeled K can be read out by executing a sequence of instructions of class T or B containing K instructions and

no shorter sequence exists to read it out. Therefore $|\text{READ}\ (R_i)| = K$.

(b) If $\ell(I_j) = 1$, I_j must have been labeled in step 3 of the labeling algorithm, and it reads out a register (explicitly or implicitly) during its execution. Therefore this register must have been labeled 1 in step 2. □

We now comment on the significance of the labels assigned by the labeling algorithm. For each register R_i, $\ell(R_i)$ indicates the minimum number of instructions of class T or B needed to read out R_i. Therefore $\ell(R_i)$ can be thought of as an "observability index" for register R_i. $\ell(I_j)$ has a similar connotation. If $\ell(I_j) \geq 2$, $\ell(I_j) - 1$ indicates the minimum number of instructions of class T or B needed to read out register $D(I_j)$. If $\ell(I_j) = 1$, instruction I_j reads out some register with label 1; thus the effects of execution of instruction I_j are directly observable at the external pins of the microprocessor if $\ell(I_j) = 1$.

Various test generation procedures to be presented in the following sections of this chapter generate tests in such a way that the knowledge gained from the correct execution of tests used to check the decoding of registers and instructions with lower labels is utilized in generating tests to check the decoding of registers and instructions with higher labels. Thus the fact that a register with a given label can be correctly "observed" is used to generate suitable tests for correctly observing registers with higher labels. Recall that a register with a lower label implies that it has better observability than the one with a higher label.

These test generation procedures may generate instructions with higher labels to set up proper operands in various registers while "checking out" instructions with lower labels. Since, as described above,

the instructions with higher labels are not checked out yet, they may not be able to set up the required operands successfully. This imposes the basic requirement on the test generation procedure: the tests must be able to check for proper execution of every instruction using other potentially faulty instructions; otherwise certain faults may mask each other and never be detected. This point will be illustrated by means of examples in Sections 4.2 and 4.3.

Since each instruction is checked for its proper execution using other potentially faulty instructions, it is not necessary to devise some labeling scheme that assigns labels to registers indicating their "distance" from the IN node which can signify their "controllability index." The test generation procedures take into consideration the presence of faulty instructions (which may fail to store required operands in registers, i.e., fail to control the registers correctly) to be used in checking out other instructions.

Recall (Example 2.4) that those instructions of class B which only change the logic level on some status pins of the microprocessor (e.g. "Interrupt enable") are not represented in the S-graph. Therefore, they are not labeled by the labeling algorithm. We assign $\ell(I_j) = 1$ for every instruction I_j of class B which is not represented in the S-graph. This assignment can be justified as follows: the effects of these instructions of class B are directly observable at the external pins of the microprocessor. Since the instructions which read out registers are labeled 1 by the labeling algorithm, it makes sense to assign label 1 to these instructions of class B.

Lemma 4.2: All instructions of class B are assigned label 1.

Proof: Those instructions of class B which are represented in the S-graph implicitly read out the program counter or a register containing the address of an instruction. These instructions are assigned label 1 in step 3 of the labeling algorithm. All the other instructions of class B (which are not represented in the S-graph) are also assigned label 1 as explained above. □

Recall that instructions with label 1 have the highest observability. Instructions of class B enjoy the highest observability.

Example 4.1: The labeling algorithm will assign the following labels to the nodes and edges of the S-graph in Figure 2.8.

Step 1: $\ell(OUT) = 0$.

Step 2: $\ell(R_1) = \ell(R_2) = \ell(R_4) = \ell(R_5) = \ell(R_6) = \ell(R_7) = 1$, $\ell(R_3) = 2$.

Step 3: $\ell(I_7) = \ell(I_8) = \ell(I_9) = \ell(I_{10}) = \ell(I_{14}) = \ell(I_{16}) = \ell(I_{17}) = \ell(I_{18}) =$
$= \ell(I_{19}) = \ell(I_{20}) = \ell(I_{21}) = 1$.

Step 4: $\ell(I_1) = \ell(I_2) = \ell(I_3) = \ell(I_4) = \ell(I_6) = \ell(I_{11}) = \ell(I_{12}) = \ell(I_{13})$
$= \ell(I_{15}) = 2$, $\ell(I_5) = 3$.

The contents of the program counter (R_6) are read out (implicitly) on the address bus during the fetching of every instruction, therefore $\ell(R_6) = 1$. The contents of the subroutine register (R_7) are implicitly read out on the address bus (by routing the contents through the program counter), hence $\ell(R_7) = 1$. Note that $\ell(I_1) = \ell(I_2) = 2$, because I_1 and I_2 both use immediate addressing (Refer to Table 2.1.), and $|D(I_1)| = |D(I_2)| = 1$. All other labels are self explanatory. □

4.2. Test Generation Procedure for Detecting Faults in the Register Decoding Function

The tests generated using the procedure guarantee that the register decoding function denoted by the mapping f_D is a one-to-one correspondence from $\Re$ to $\Re$.

This procedure uses two data structures, a queue of registers and a set of registers which are named Q and A respectively. The queue Q is initialized with all the registers such that a register R_i lies ahead of another register R_j in the queue, if and only if, $\ell(R_i) \leq \ell(R_j)$. The set A is initialized to be empty. In each iteration of the test generation procedure, set A is progressively augmented by removing the register lying in the front of the queue Q and including it in set A; now the register which was second in the queue before the augmentation of set A lies in the front of the queue, i.e., the queue is updated. The tests generated so far will assure that at any given stage, registers in set A have disjoint image sets under mapping f_D. The procedure terminates when set A contains all the registers that were initially in the queue and the queue gets empty. At this stage, the generated tests will guarantee that all the registers have disjoint image sets under mapping f_D, establishing that f_D is a one-to-one correspondence. Recalling the discussion in Section 4.1, the procedure utilizes the knowledge gained from the correct execution of tests used to check decoding of registers with lower labels to generate tests to check decoding of registers with higher labels.

ONE and ZERO will be frequently used as operands or addresses of operands in various test procedures. This choice is arbitrary. We could have used (1010...10) and its bit-wise complement (0101...01) as operands or addresses of operands instead.

Procedure 4.1 given below generates tests for detecting faults in the register decoding function. Note that the test instructions are generated only in steps 3(a), (b), and (c); other steps perform bookkeeping tasks.

Procedure 4.1: Procedure to generate tests for detecting faults in the register decoding function

Step 1: Initialize the queue Q with all the registers such that register R_i lies ahead of register R_j, if and only if, $\ell(R_i) \leq \ell(R_j)$; Initialize the set A as empty.

Step 2: A ← register at the front of Q; Update Q.

Step 3: REPEAT

a) Generate instructions to write each register R_i of set A with data ONE, and the register at the front of Q (if there is one) with data ZERO. (The instructions of the corresponding WRITE (R_i) sequences need to be generated.)

b) Generate instructions to read out each register R_i of set A, such that register R_i will be read before register R_j, if and only if, $\ell(R_i) \leq \ell(R_j)$. (The instructions of the corresponding READ (R_i) sequences need to be generated.)

c) Generate instructions to read out the register R_j at the front of Q (if there is one). (The instructions of the READ (R_j) sequence need to be generated.)

d) A ← A ∪ {Register at the front of Q}.

e) Update Q.

UNTIL Q = empty.

Step 4: Repeat steps 1, 2, and 3 with complementary data.

Procedure 4.1 describes the test generation procedure at a higher level. There are many subtle points involved in the execution details of this procedure, particularly if some registers can be written or read out only implicitly. These points can be best illustrated by giving an example. We show how to apply this algorithm to generate the tests for the S-graph of Figure 2.8 in the following example accompanied with the explanatory comments.

Example 4.2: Generation of tests for detecting faults in the register decoding function for the S-graph of Figure 2.8.

Step 1: $Q \leftarrow R_1R_2R_4R_5R_6R_7R_3$; $A \leftarrow \emptyset$

Step 2: $A \leftarrow \{R_1\}$; $Q \leftarrow R_2R_4R_5R_6R_7R_3$

Step 3:

Iteration 1

a) I_1 with operand ONE; I_2 with operand ZERO;

b) I_7; /Expected output data = ONE/

c) I_8; /Expected output data = ZERO/

d) $A \leftarrow \{R_1, R_2\}$

e) $Q \leftarrow R_4R_5R_6R_7R_3$

Iteration 2

a) I_1 with operand ONE; I_2 with operand ONE;
I_{15} with operand ZERO; /stack pointer (R_4) is written with a ZERO/

b) I_7; I_8; /R_1 and R_2 are read out; expected output data = ONE/

c) I_{16}; /stack pointer is implicitly "read out" on the address bus; expected output "data" = ZERO/

d) $A \leftarrow \{R_1, R_2, R_4\}$

e) $Q \leftarrow R_5 R_6 R_7 R_3$

Iteration 3

a) I_1 with operand ONE; I_2 with operand ONE;
I_{15} with operand ONE; /R_1, R_2, stack pointer (R_4) are written with data ONE/
I_{17} with address of the operand ZERO; /R_5 is written implicitly with "data" ZERO/

b) I_7; I_8; I_{16}; /R_1 and R_2 are explicitly read out while the stack pointer (R_4) is implicitly read out/

c) I_{17} with the address of operand ZERO; /R_{15} is implicitly read out on the address bus with expected output "data" = ZERO/

d) $A \leftarrow \{R_1, R_2, R_4, R_5\}$

e) $Q \leftarrow R_6, R_7, R_3$

Iteration 4

a) I_1 with operand ONE; I_2 with operand ONE;
I_{15} with operand ONE; /R_1, R_2, stuck pointer (R_4) are written with data ONE/
I_{17} with the address of operand ONE; /R_5 is written implicitly with "data" ONE/
I_9 with jump address = LOC 1; /program counter is written implictily with data = LOC 1. LOC 1 is chosen different from ONE/

b,c) LOC 1: I_7; /I_7 is stored in location LOC 1. When this instruction is fetched, the program counter is implicitly read out on the address bus; expected output "data" = LOC 1 ≠ ONE. R_1 is explicitly read out as I_7 is executed/

I_8; I_{16}; I_{17} with the address of operand ONE; /R_2 is explicitly read out. R_4 and R_5 are implicitly read out/

d) $A \leftarrow \{R_1, R_2, R_4, R_5, R_6\}$

e) $Q \leftarrow R_7, R_3$

Iteration 5

a) I_1 with operand ONE; I_2 with operand ONE;

I_{15} with operand ONE; /R_1, R_2, stack pointer (R_4) are written with data ONE/

I_{17} with the address of operand ONE; /R_5 is written implicitly with "data" ONE/

I_9 with jump address = LOC 2; /program counter is written implicitly with data = LOC 2/

LOC 2: I_{20} with jump address = LOC 3; /Note that I_{20} is the "Jump to subroutine" instruction, hence program counter (now containing LOC 2 + 1) is saved in the subroutine register (R_7), and a new jump address = LOC 3 is loaded into the program counter. Thus R_6 and R_7 are written implicitly with "data" LOC 3 and LOC 2 + 1, respectively. Choose LOC 3 different from LOC 2 + 1 and also different from ONE/

b,c) LOC 3: I_7; $/I_7$ is stored in LOC 3. When I_7 is fetched, the program counter is read out implicitly on the address bus; expected output "data" = LOC 3. Thus, I_7 is the first instruction in the subroutine. I_7 explicitly reads out R_1/

I_8; I_{16}; I_{17} with the address of operand ONE; $/R_2$ is explicitly read out. R_4 and R_5 are implicitly read out/

I_{21}; $/I_{21}$ is the "Return from subroutine" instruction. The contents of the subroutine register (R_7) are transferred to the program counter. The next instruction will be fetched from the location LOC 2 + 1, as LOC 2 + 1 is the return address for the subroutine. When this new instruction is fetched, the subroutine register will be effectively read out through the program counter/

d) $A \leftarrow \{R_1, R_2, R_4, R_5, R_6, R_7\}$

e) $Q \leftarrow R_3$

Iteration 6

a) I_1 with operand ZERO; $/I_1$ is stored in location LOC 2 + 1.
[illegible] is fetched R_7 is implicitly read out as explained

[illegible] with data ZERO/

I [illegible] ONE; I_2 with operand ONE;

I_{15} with operand ONE; I_{17} with the address of operand ONE;

$/R_1$, R_2, R_4, and R_5 are written with "data" ONE/

I_9 with jump address = LOC 4; /the program counter implicitly written with LOC 4. Choose LOC 4 different from ZERO/

LOC 4: I_{20} with jump address = LOC 5; /the subroutine register (R_7) is written implicitly with data LOC 4 + 1. Choose LOC 4 + 1 different from ZERO. When I_{20} is fetched the program counter is implicitly read out; expected output data = LOC 4/

b) LOC 5: I_7; I_8; I_{16}; I_{17} with the address of operand ONE; /R_1 and R_2 are explicitly read out; R_4 and R_5 are implicitly read out/

I_{21}; /causes the subroutine register (R_7) to be implicitly read out through the program counter when the next instruction will be fetched from location LOC 4 + 1

c) LOC 4 + 1. I_6; I_7; /R_3 is read out using READ (R_3) = $\langle I_6, I_7 \rangle$ sequence; expected output data = ZERO. When I_6 is fetched R_7 is implicitly read out as explained above/

d) $A \leftarrow \{R_1, R_2, R_4, R_5, R_6, R_7, R_3\}$

e) $Q \leftarrow$ empty

Step 4: Repeat steps 1, 2, 3 using complementary data. □

The generated test sequence will be I_1, I_2, I_7, I_8, I_1, I_2, I_{15}, ..., I_7, I_8, I_{16}, I_{17}, I_{21}, I_6, I_7. Thus the "writing" process involved in step 3(a), and the "reading" process involved in step 3(b) and (c), do involve implicit writing and reading of registers. The whole procedure requires careful selection of "data", i.e., both the operands and address of operands. The jump addresses in the "Jump", "Jump to

subroutine", and "Return from subroutine" instructions must be carefully chosen to avoid reexecuting the already executed tests or overwriting the instructions.

We now present a lemma describing the behavior of a microprocessor under faulty register decoding. This will then be used to prove that the tests generated using Procedure 4.1 will detect any detectable fault in the register decoding function.

Lemma 4.3: Let READ $(R_i) = \langle I_{k_1}, I_{k_2}, \ldots, I_{k_m} \rangle$, and WRITE $(R_i) = \langle I_{p_1}, I_{p_2}, \ldots, I_{p_n} \rangle$.

a) When R_i is written with data d by executing the instructions in the WRITE (R_i) sequence, all registers in the set $f_D(D(\text{WRITE } (R_i))) = f_D(D(I_{p_1}, I_{p_2}, \ldots, I_{p_n}))$ are written with data d, unless (1). $f_D(D(I_{p_j})) = \{\phi\}$ for some $I_{p_j} \in$ WRITE (R_i), $1 \leq j \leq n-1$, in which case R_i is written with either a ONE or a ZERO depending on technology, or (2). $f_D(D(I_{p_n})) = \{\phi\}$, in which case the contents of R_i remain unchanged.

b) When R_i is read out by executing the instructions in the READ (R_i) sequence, during the process of reading out R_i all registers in the set $f_D(D(I_{k_1}, I_{k_2}, \ldots, I_{k_{m-1}}))$ are written with data d and data d is read out, if R_i contains data d, unless $f_D(S(I_{k_j})) = \{\phi\}$, for some $I_{k_j} \in$ READ (R_i), in which case a ONE or a ZERO will be read out, depending on technology.

Proof: The lemma follows immediately from the faulty behavior of the register decoding function f_D. □

Example 4.3: With reference to the S-graph of Figure 2.8, READ $(R_3) = \langle I_6, I_7 \rangle$ and WRITE $(R_3) = \langle I_1, I_5 \rangle$. When R_3 is written with data d by executing I_1 with data d, and I_5, all registers in the set

$f_D(D(WRITE(R_3))) = f_D(D(I_1, I_5)) = f_D(D(I_1)) \cup f_D(D(I_5)) = f_D(R_1) \cup f_D(R_3)$ are written with data d, unless $f_D(R_1) = \{\phi\}$, in which case R_3 is written with either a ONE or a ZERO, or $f_D(R_3) = \{\phi\}$, in which case the contents of R_3 remain unchanged. When R_3 is read out by executing I_6 and I_7, all registers in the set $f_D(R_1)$ are written with data d and data d is read out, unless $f_D(R_1) = \{\phi\}$ or $f_D(R_3) = \{\phi\}$, in which case either a ONE or a ZERO is read out. □

Theorem 4.1: The test sequence generated by using Procedure 4.1 is capable of detecting any detectable fault in the fault model for the register decoding function.

Proof: The proof is by induction. At the beginning of step 3(a) of Procedure 4.1, let set $A = \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}$. Let the induction hypothesis be $f_D(R_{i_1}) \cap f_D(R_{i_2}) \cap \ldots \cap f_D(R_{i_k}) = \{\phi\}$, and $f_D(R_{i_j}) \neq \{\phi\}$, for each $R_{i_j} \in A$. At the end of step 3(d) of Procedure 4.1, i.e., when set A is augmented, let set $A = \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}, R_{i_{k+1}}\}$. We will prove that $f_D(R_{i_1}) \cap f_D(R_{i_2}) \cap \ldots\ldots \cap f_D(R_{i_k}) \cap f_D(R_{i_{k+1}}) = \{\phi\}$, and $f_D(R_{i_{k+1}}) \neq \{\phi\}$.

In step 3(a), registers of set A are written with data ONE (ZERO), and register $R_{i_{k+1}}$, which is at the front of the queue, is written with data ZERO (ONE), by executing* the instructions in the corresponding WRITE sequences. If $f_D(R_{i_{k+1}}) = \{\phi\}$, the fault will be readily detected when appropriate instructions are executed to read out $R_{i_{k+1}}$ in step 3(c), as it will fail to produce either a ONE or a ZERO following Lemma 4.3. Assume that for some register R_{i_ℓ} of set A, $f_D(R_{i_\ell}) \cap f_D(R_{i_{k+1}}) \neq \{\phi\}$. If R_{i_ℓ} is written after $R_{i_{k+1}}$ in step 3(a), the fault will be detected when

*Though strictly speaking Procedure 4.1 is a test generation procedure in the proof we are assuming that the tests are executed.

$R_{i_{k+1}}$ is read out in step 3(c), since according to Lemma 4.3, all the registers in the set $f_D(R_{i_\ell})$, and hence in the set $f_D(R_{i_\ell}) \cap f_D(R_{i_{k+1}})$ will be written with ONE (ZERO). Since $f_D(R_{i_\ell}) \cap f_D(R_{i_{k+1}}) \subseteq f_D(R_{i_{k+1}})$, when $R_{i_{k+1}}$ is read out in step 3(c), it will either produce a ONE instead of a ZERO, or a ZERO instead of a ONE. Note that since $\ell(R_{i_\ell}) \leq \ell(R_{i_{k+1}})$, the process of reading out of R_{i_ℓ} will not require routing of R_{i_ℓ} through $R_{i_{k+1}}$. Similar arguments apply when R_{i_ℓ} is written before $R_{i_{k+1}}$ in step 3(a). In this case the fault will be detected when R_{i_ℓ} is read out in step 3(c).

The basis of induction, i.e., when $A = \{R_{i_1}\}$, $f_D(R_{i_1}) \neq \{\phi\}$, and when $A = \{R_{i_1}, R_{i_2}\}$, $f_D(R_{i_1}) \cap f_D(R_{i_2}) = \{\phi\}$ can be readily proved following the same arguments used so far. Using these arguments, it is guaranteed that $\bigcap_{i=1}^{n} f_D(R_i) = \{\phi\}$, where $\mathfrak{R} = \{R_1, R_2, \ldots, R_n\}$. Since all the registers have disjoint image sets under mapping f_D, f_D cannot be a many-to-one correspondence. Moreover, since $f_D(R_i) \neq \{\phi\}$ for each $R_i \in \mathfrak{R}$, f_D cannot be a one-to-many correspondence from $\mathfrak{R}$ to $\mathfrak{R}$. Therefore f_D is guaranteed to be a one-to-one correspondence from $\mathfrak{R}$ to $\mathfrak{R}$. If follows immediately that the register decoding function (denoted by f_D) is free of any detectable fault. Note that for some registers, even if $f_D(R_i) \neq \{R_i\}$, f_D could still be a one-to-one correspondence. For example, we may have $f_D(R_i) = \{R_j\}$ and $f_D(R_j) = \{R_i\}$. In such a case, the fault is an undetectable fault. □

4.3. Test Generation Procedures for Detecting Faults in the Instruction Decoding and Control Function

In this section we present the test generation procedures to detect faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$ for each ordered pair of instructions I_j and I_k. We divide the overall task of test generation into three subtasks depending on which class (T, M, or B) instructions I_j and I_k belong to. Following this, we first give the order in which the tests are applied, and then describe the details of test generation.

The overall task of detecting faults in the instruction decoding and control function can be divided into three subtasks.

subtask 1: Test for fault $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$, where $I_j \in$ class T, and $I_k \in$ class $(T \cup M)$.

subtask 2: Test for faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$, where $I_j \in$ class M, and $I_k \in$ class $(T \cup M)$.

subtask 3: Test for faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$, where $I_j \in$ class B, and $I_k \in$ class $(T \cup M \cup B)$; or $I_j \in$ class $(T \cup M)$ and $I_k \in$ class B.

The basic philosophy behind this task division is to employ a systematic approach that tackles a complex problem by dividing it into logically distinct and smaller subproblems.

4.3.1. Order of Test Application

Before presenting the details of test generation, we first describe the order in which the tests are applied. The tests for each subtask described above are to be executed by the microprocessor under test in the order given below.

First we concentrate on instructions with label 1, i.e., $\ell(I_j) = \ell(I_k) = 1$, and apply tests to detect faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$. Then we apply tests to detect faults $f(I_j/I_j+I_k)$, where $\ell(I_j) = 1$ and $\ell(I_k) = 2$. This is followed by tests to detect faults $f(I_j/I_j+I_k)$, where $2 \leq \ell(I_j) \leq K_{max}$ and $\ell(I_k) = 1$. (K_{max} indicates the maximum value of the labels of edges representing instructions in the S-graph.) Thus we check that all instructions with label 1 are decoded correctly, no instruction with label 1 causes additional execution of an instruction with label 2, and no instruction with a label greater than 1 causes additional execution of an instruction with label 1. This procedure can be easily generalized and is given in a precise algorithmic form below.

Algorithm 4.2: Algorithm to determine the order of test application for detecting $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$

FOR $K \leftarrow 1$ TO K_{max} DO

BEGIN

Step 1: Apply tests to detect faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$, where $\ell(I_j) = \ell(I_k) = K$.

Step 2: Apply tests to detect faults $f(I_j/I_j+I_k)$, where $1 \leq \ell(I_j) \leq K$, $\ell(I_k) = K + 1$, and $K < K_{max}$.

Step 3: Apply tests to detect faults $f(I_j/I_j+I_k)$, where $K + 1 \leq \ell(I_j) \leq K_{max}$, $\ell(I_k) = K$.

END

Strictly speaking, this order need not be followed during the actual application of tests, but it plays a very crucial role in

proving that the tests detect faults in the instruction decoding and control function. Therefore we assume that the tests are applied in the order given by Algorithm 4.2.

Note that in steps 2 and 3 it is not necessary to check for faults $f(I_j/\phi)$ or $f(I_j/I_k)$ because these faults are detected by tests involved in step 1. We now present an example to illustrate the three steps of the algorithm.

For concise representation we introduce some notation at this stage. Let I_A, I_B, etc., denote sets of instructions. Then $f(I_A/I_B)$ would denote a set of faults given by $\{f(I_i/I_j) \mid I_i \in I_A \text{ and } I_j \in I_B\}$. For example, let $I_A = \{I_1, I_2\}$ and $I_B = \{I_3, I_4\}$, then $f(I_A/I_B) = \{f(I_1/I_3), f(I_1/I_4), f(I_2/I_3), f(I_2/I_4)\}$. Similarly $f(I_A/I_A+I_B)$ and $f(I_A/\phi)$ denote the corresponding sets of faults. Needless to say, we do not incorporate $f(I_i/I_i)$ or $f(I_i/I_i+I_i)$ in the sets $f(I_A/I_B)$ or $f(I_A/I_A+I_B)$. $\ell(I_A)$ denotes the set of labels of instructions in set I_A.

Example 4.4: This example illustrates the steps of Algorithm 4.2 in the case of subtask 1 for the S-graph of Figure 2.8.

Iteration 1

Step 1: Apply tests for faults $f(I_A/\phi)$, $f(I_A/I_B)$, and $f(I_A/I_A+I_B)$ where $I_A = I_B = \{I_7, I_8, I_{16}, I_{17}, I_{18}, I_{19}\}$, and $\ell(I_A) = \ell(I_B) = \{1\}$.

Step 2: Apply tests for faults $f(I_A/I_A+I_B)$ where $I_A = \{I_7, I_8, I_{16}, I_{17}, I_{18}, I_{19}\}$, $I_B = \{I_1, I_2, I_3, I_4, I_6, I_{11}, I_{12}, I_{13}, I_{15}\}$, and $\ell(I_A) = \{1\}$, $\ell(I_B) = \{2\}$.

Step 3: Apply tests for faults $f(I_A/I_A+I_B)$ where $I_A = \{I_1, I_2, I_3, I_5, I_6, I_{15}\}$, $I_B = \{I_7, I_8, I_{16}, I_{17}, I_{18}, I_{19}\}$, and

$\ell(I_A) = \{2, 3\}$, $\ell(I_B) = \{1\}$.

Iteration 2:

Step 1: Apply tests for faults $f(I_A/\phi)$, $f(I_A/I_B)$, and $f(I_A/I_A+I_B)$ where $I_A = \{I_1, I_2, I_3, I_6, I_{15}\}$, $I_B = \{I_1, I_2, I_3, I_4, I_6, I_{11}, I_{12}, I_{13}, I_{15}\}$, and $\ell(I_A) = \ell(I_B) = 2$.

Step 2: Apply tests for faults $f(I_A/I_A+I_B)$ where $I_A = \{I_7, I_8, I_{16}, I_{17}, I_{18}, I_{19}, I_1, I_2, I_3, I_6, I_{15}\}$, $I_B = \{I_5\}$, and $\ell(I_A) = \{1, 2\}$, $\ell(I_B) = \{3\}$.

Step 3: Apply tests for faults $f(I_A/I_A+I_B)$ where $I_A = \{I_5\}$, $I_B = \{I_1, I_2, I_3, I_4, I_6, I_{11}, I_{12}, I_{13}, I_{15}\}$, and $\ell(I_A) = \{3\}$, $\ell(I_B) = \{2\}$.

Iteration 3:

Step 1: Apply tests for fault $f(I_5/\phi)$. (Note that $\ell(I_5) = \{3\}$.) □

The next job is to develop the details of test generation for detecting faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$. These details depend very heavily on the labels of instructions and their source and destination registers. Therefore we partition the job into various cases depending on these labels and present the test generation procedure for each case.

At this stage we make an assumption about the labels of edges representing instructions. If $I_j \in$ class M then $\ell(I_j) \leq 2$, i.e., if $\ell(I_j) > 2$ then $I_j \in$ class T. Recall from Lemma 4.2 that all instructions of class B are labeled 1. Thus the destination of an instruction of class M can only be a main memory location or a register with label 1. This assumption can be easily justified for available microprocessors,

since the result of an instruction of class M is usually stored in a main memory location or an accumulator [Cush77]; on the other hand, the destination register of an instruction of class T can have any label. Without this assumption the details of the test generation procedures become extremely complicated.

4.3.2. Test Generation for $f(I_j/\phi)$

The details of test generation depend principally on $\ell(I_j)$. We consider three cases, namely, case A(1), case A(2), and case A(3), depending on $\ell(I_j)$. The suffix A is used to denote that a case belongs to the details of test generation for $f(I_j/\phi)$. These cases are divided into subcases which are listed in Table 4.1. For each subcase, the table gives which test generation procedure is applicable and which theorem proves the fault coverage.

4.3.2.1. Test Generation for $f(I_j/\phi)$ When $\ell(I_j) = 1$

This case is referred to as case A(1) and is divided into two subcases.

Case A(1.1): OUT $\in D(I_j)$, i.e., I_j is expected to read out a register with label 1 (according to Lemma 4.1).

Case A(1.2): I_j is an instruction of class B not represented in the S-graph, i.e., I_j only changes the logic level on some status pins.

In either subcase the fault detection is easy since I_j has the highest observability as signified by $\ell(I_j) = 1$.

We give the test generation procedures below. It is straightforward to derive the tests in terms of the assembly language instructions of the microprocessor to be tested from the procedures to follow. For

Table 4.1. Cases for the details of test generation for detecting $f(I_j/\phi)$.

Case and its description		Test generation procedure	Theorem proving the fault coverage
Case A(1) $\ell(I_j) = 1$	Case A(1.1) OUT $\in D(I_j)$	Procedure 4.2	*
	Case A(1.2) $I_j \in$ class B, and it is not represented in the S-graph.	Procedure 4.3	*
Case A(2) $\ell(I_j) = 2$		Procedure 4.4	Theorem 4.2
Case A(3) $\ell(I_j) = K \geq 3$	Case A(3.1) $S(I_j)$ is not the destination register of any instruction belonging to the READ $(D(I_j))$ sequence.	Procedure 4.4	Theorem 4.3
	Case A(3.2) $S(I_j)$ is the destination register of an instruction belonging to the READ $(D(I_j))$ sequence.	Procedure 4.5	Theorem 4.4

*Fault coverage follows immediately from the description of the procedure.

conciseness we say that the microprocessor executes these procedures instead of saying that the microprocessor executes the tests generated by these procedures. For the same reason we treat these procedures as if they are test execution procedures instead of test generation procedures. We denote various operands for instructions by OPERAND 1, OPERAND 2, etc., and various results stored in registers (as a consequence of instruction execution) by RESULT 1, RESULT 2, etc.

Procedure 4.2:

This procedure is applicable for case A(1.1). It generates tests to detect fault $f(I_j/\phi)$ when $\ell(I_j) = 1$ and OUT $\in D(I_j)$.

Step 1: Store proper operand(s) in $S(I_j)$ such that when I_j is executed, the expected output "data" is different from the quiescent logic value on the data (or address) bus.

Step 2: Execute I_j. □

Procedure 4.3:

This procedure is applicable for case A(1.2). It generates tests to detect fault $f(I_j/\phi)$ when $\ell(I_j) = 1$ and I_j belongs to class B but it is not represented in the S graph.

Step 1: Execute the proper instruction to set up the logic value on a status pin to x ($x \in \{0, 1\}$) if the instruction I_j under consideration, when executed, sets up the logic value on that status pin to $\bar{x}$.

Step 2: Execute I_j. □

4.3.2.2. Test Generation for $f(I_j/\phi)$ When $\ell(I_j) = 2$

This case is referred to as case A(2). In this case $|READ\ (D(I_j))| = 1$. Let $READ\ (D(I_j)) = \langle I_k \rangle$. Of course, $\ell(I_k) = 1$, and by definition of the READ sequence $I_k \in (T \cup B)$ and $OUT \in D(I_k)$.

Procedure 4.4:

This procedure is applicable for case A(2). It generates tests to detect fault $f(I_j/\phi)$ when $\ell(I_j) = 2$.

Step 1: Store OPERAND 1 in $D(I_j)$ and proper operand(s) in $S(I_j)$ such that when I_j is executed it produces RESULT 1 in $D(I_j)$, and RESULT 1 $\neq$ OPERAND 1.

Step 2: Read out $D(I_j)$ by executing $READ\ (D(I_j))$.
/Expected output data = OPERAND 1/

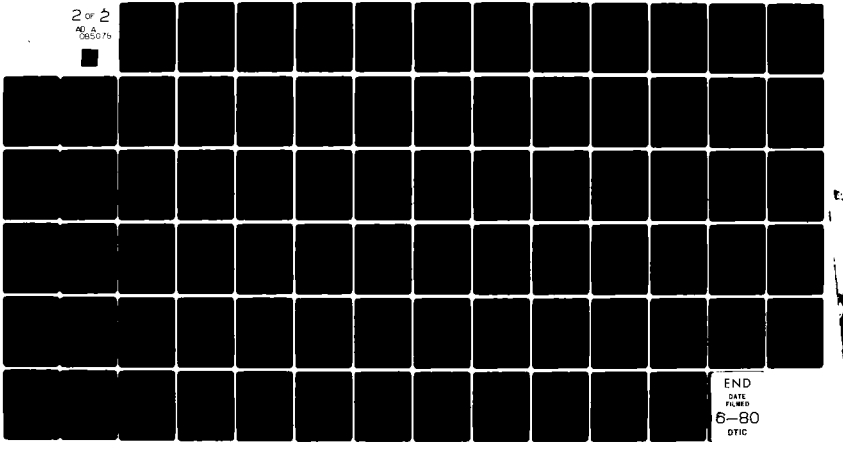
Step 3: Execute I_j.

Step 4: Read out $D(I_j)$ by executing $READ\ (D(I_j))$.
/Expected output data = RESULT 1/ □

Example 4.5: This example (depicted in Figure 4.1) illustrates Procedure 4.4. I_j is an "Add" instruction which adds the contents of registers R_1 and R_2 and stores the result in R_3. $READ\ (D(I_j)) = \langle I_k \rangle$, $\ell(I_k) = 1$ and $I_k \in$ class T. OPERAND 1 can be chosen to be ONE, and RESULT 1 to be ZERO, requiring that operand ZERO be stored in both R_1 and R_2. I_k is executed to make sure that R_3 does store OPERAND 1. This is followed by execution of I_j and I_k. □

AD-A085 078 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
TEST GENERATION FOR MICROPROCESSORS.(U)
MAY 79 S M THATTE N00014-79-C-0424
UNCLASSIFIED R-842 NL

2 OF 2
AD A 085078



END
DATE FILMED
6-80
DTIC

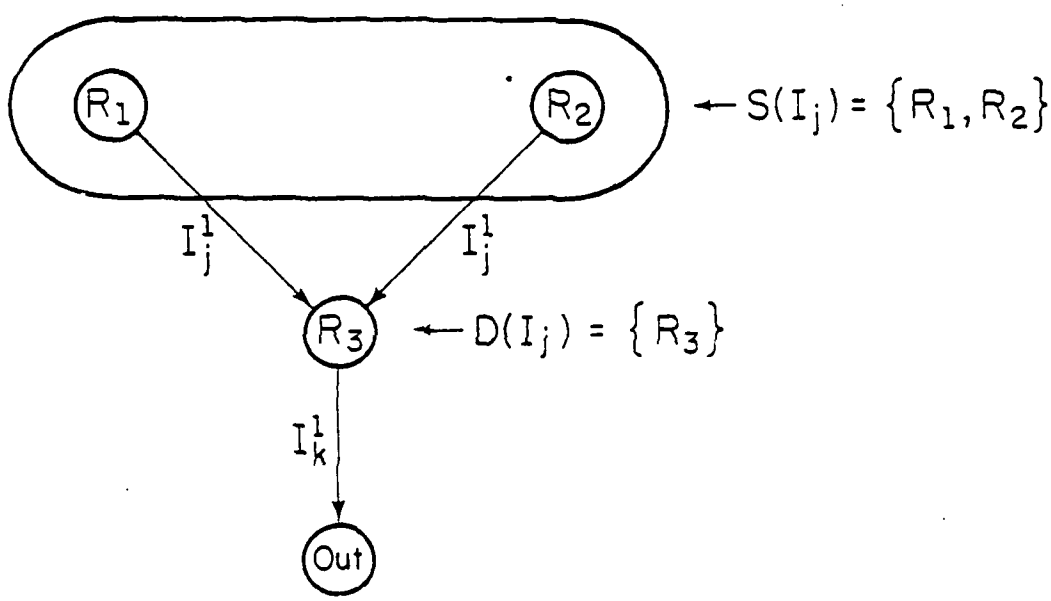


Figure 4.1. Example illustrating Procedure 4.4.

Theorem 4.2: Procedure 4.4 detects $f(I_j/\phi)$ in case A(2).

Proof: Note that $\ell(I_k) = 1$ where READ $(D(I_j)) = \langle I_k \rangle$. (Refer to Figure 4.1.) Since the tests are applied in the order specified by Algorithm 4.2, the microprocessor under test executes this procedure after executing the tests required to detect $f(I_p/\phi)$, $f(I_p/I_q)$, and $f(I_p/I_p+I_q)$ where $\ell(I_p) = \ell(I_q) = 1$, and $f(I_v/I_v+I_w)$ where $\ell(I_v) = 1$ and $\ell(I_w) = 2$. Therefore, when the microprocessor under test executes this procedure it has been already checked that the READ $(D(I_j)) = \langle I_k \rangle$ sequence can correctly read out $D(I_j)$ and its execution does not cause additional execution of any instruction with label 2; in particular the contents of $D(I_j)$ are not changed after the execution of READ $(D(I_j))$. Therefore step 2 ensures that $D(I_j)$ stores OPERAND 1. This step is necessary because due to faults involved in the instructions used to write data in $D(I_j)$, RESULT 1 may be stored in $D(I_j)$ instead of OPERAND 1. If step 2 is not executed, the fault $f(I_j/\phi)$ will be masked.

In step 3, I_j is executed which is expected to produce RESULT 1 in $D(I_j)$. If $f(I_j/\phi)$ is present, the contents of $D(I_j)$ will not change. Consequently when $D(I_j)$ is read out in step 4 the fault will be detected.

4.3.2.3. Test Generation for $f(I_j/\phi)$ When $\ell(I_j) = K \geq 3$

This case is referred to as case A(3). According to our assumption in Section 4.3.1, $I_j \in$ class T. This case is divided into two subcases.

Case A(3.1): $S(I_j)$ is not the destination register of any instruction belonging to the READ $(D(I_j))$ sequence. Figure 4.2 illustrates this case, $\ell(I_j) = K \geq 3$, $\ell(D(I_j)) = K-1$, $\ell(S(I_j)) = K$, and READ $(D(I_j)) = \langle I_{p_{K-1}}, I_{p_{K-2}}, \ldots, I_{p_1} \rangle$. Thus it is possible to read out $D(I_j)$ without routing the contents of $D(I_j)$ through $S(I_j)$. In this case we follow Procedure 4.4. (The same procedure which is used in case A(2).)

Theorem 4.3: Procedure 4.4 detects $f(I_j/\phi)$ in case A(3.1).

Proof: The proof follows the same arguments given for the proof of Theorem 4.2, except for one change. The microprocessor under test executes this procedure after executing the tests required to detect $f(I_p/\phi)$, $f(I_p/I_q)$, $f(I_p/I_p+I_q)$ where $1 \leq \ell(I_p), \ell(I_q) \leq K-1$, and $f(I_v/I_v+I_w)$ where $1 \leq \ell(I_v) \leq K-1$ and $\ell(I_w) = K$. Note that in this case READ $(D(I_j)) = \langle I_{p_{K-1}}, I_{p_{K-2}}, \ldots, I_{p_1} \rangle$ as illustrated in Figure 4.2. Since $\ell(D(I_j)) = K-1$, $\ell(I_{p_i}) = i$, for $1 \leq i \leq K-1$.

Therefore, when the microprocessor under test executes this procedure it has been already checked that the execution of any instruction $I_{p_i} \in$ READ $(D(I_j))$ does not give rise to the additional execution of any instruction with label K. $D(I_j)$ can be correctly read out by executing READ $(D(I_j))$ in step 2 of this procedure, and after the execution of READ $(D(I_j))$ the contents of $D(I_j)$ are not changed (note that $\ell(D(I_j)) = K-1$ and $\ell(I_j) = K$). All the remaining arguments are exactly the same as given in the proof of Theorem 4.2. □

Case A(3.2): $S(I_j)$ is the destination register of an instruction belonging to the READ $(D(I_j))$ sequence. Figure 4.3 illustrates this case; $\ell(I_j) = K \geq 3$, $\ell(D(I_j)) = K-1$, $\ell(S(I_j)) = K-2$.

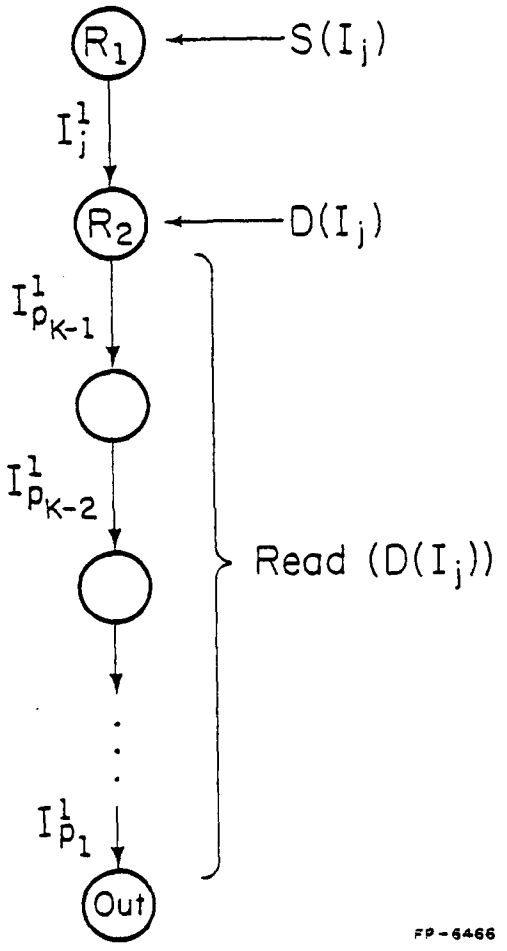


Figure 4.2. Illustrating case A(3.1) in Section 4.3.2.

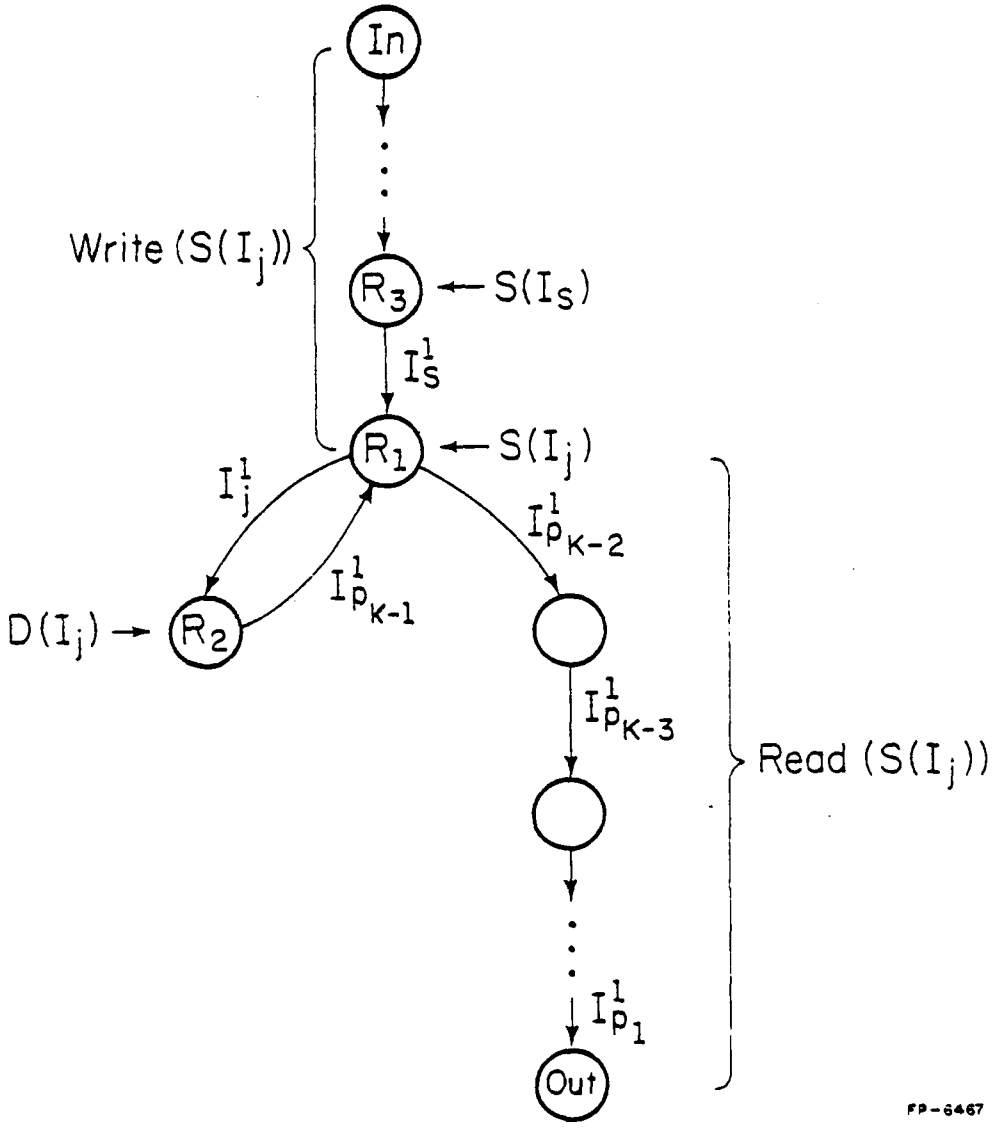


Figure 4.3. Illustrating case A(3.2) in Section 4.3.2.

READ $(D(I_j))$ = $\langle I_{p_{K-1}}, I_{p_{K-2}}, \ldots, I_{p_1} \rangle$ and READ $(S(I_j))$ = $\langle I_{p_{K-2}}, I_{p_{K-3}}, \ldots, I_{p_1} \rangle$ where $\ell(I_{p_i}) = i$ for $1 \leq i \leq K-1$. Let $I_s \in$ WRITE$(S(I_j))$ and $D(I_s) = S(I_j)$. Therefore, $\ell(I_s) = K-1$, and $\ell(S(I_s)) = K-1$.

Procedure 4.5:

This procedure is applicable for case A(3.2). It generates tests to detect fault $f(I_j/\phi)$ when $\ell(I_j) = K \geq 3$, and $S(I_j)$ is the destination register of an instruction belonging to the READ $(D(I_j))$ sequence.

Step 1: Store OPERAND 1 in $D(I_j)$ by executing WRITE $(D(I_j))$.

Step 2: Read out $D(I_j)$ by executing READ $(D(I_j))$.

/Expected output data = OPERAND 1/

Step 3: Store OPERAND 2 $\neq$ OPERAND 1 in $S(I_s)$ by executing WRITE $(S(I_s))$.

Step 4: Read out $D(I_j)$ by executing READ $(D(I_j))$.

/Expected output data = OPERAND 1/

Step 5: Execute I_s and I_j. /Refer to Figure 4.3/

Step 6: Read out $D(I_j)$ by executing READ $(D(I_j))$.

/Expected output data = OPERAND 2 $\neq$ OPERAND 1/ □

Theorem 4.4: Procedure 4.5 detects $f(I_j/\phi)$ in case A(3.2).

Proof: $1 \leq \ell(I_{p_i}) = i \leq K-1$, for each instruction I_{p_i} in the READ $(D(I_j))$ sequence. When the microprocessor under test executes this procedure it has already executed the tests required to detect $f(I_p/\phi)$, $f(I_p/I_q)$ and $f(I_p/I_p+I_q)$ where $1 \leq \ell(I_p), \ell(I_q) \leq K-1$, and $f(I_v/I_v+I_w)$ where $1 \leq \ell(I_v) \leq K-1$ and $\ell(I_w) = K$.

Therefore, when the microprocessor under test executes this procedure it has been already checked that the READ ($D(I_j)$) sequence can correctly read out $D(I_j)$ and the execution of any instruction in this sequence will not give rise to additional execution of any instruction with label K; in particular the contents of $D(I_j)$ and $S(I_s)$ remain unchanged. (Note that $\ell(D(I_j)) = \ell(S(I_s)) = K-1$.) Moreover, the execution of I_s does not give rise to additional execution of any instruction with label K, since $\ell(I_s) = K-1$; thus in particular the contents of $D(I_j)$ remain unchanged after the execution of I_s.

In step 2, $D(I_j)$ is correctly read out by executing READ ($D(I_j)$) and it continues to store the operand stored in it, i.e., OPERAND 1, after this "read out" process. In step 3, $S(I_s)$ is written with OPERAND 2 by executing WRITE ($S(I_s)$). Since $\ell(S(I_s)) = \ell(D(I_j)) = K-1$, it is possible to write data in $S(I_s)$ without routing it through $D(I_j)$ during the execution of WRITE ($S(I_s)$). Step 4 ensures that $D(I_j)$ continues to store OPERAND 1 after step 3. Moreover, $D(I_j)$ and $S(I_s)$ continue to store OPERAND 1 and OPERAND 2 , respectively, after the "read out" process in step 4.

Since $\ell(I_s) = K-1$, the execution of I_s in step 5 will correctly transfer the contents of $S(I_s)$ to $S(I_j)$, i.e., $S(I_j)$ now contains OPERAND 2 $\neq$ OPERAND 1. Also the contents of $D(I_j)$ remain unchanged, since $\ell(I_s) = K-1$ and $\ell(D(I_j)) = K-1$. After this I_j is executed which is expected to change the contents of $D(I_j)$ to OPERAND 2. If $f(I_j/\phi)$ exists, I_j will fail to do so and the fault will be detected when $D(I_j)$ is read out in step 6. □

4.3.3. Test Generation for $f(I_j/I_k)$

The details of test generation depend principally on $\ell(I_j)$ and $\ell(I_k)$. We consider three cases, namely, case B(1), case B(2), and case B(3) depending on $\ell(I_j)$. The suffix B is used to denote that a case belongs to the details of test generation for $f(I_j/I_k)$. These cases are divided into subcases which are listed in Table 4.2. For each case, the table gives which test generation procedure is applicable and which theorem proves the fault coverage.

4.3.3.1. Test Generation for $f(I_j/I_k)$ when $\ell(I_j) = 1$

This case is referred to as case B(1) and is divided into two subcases depending on $\ell(I_k)$.

Case B(1.1): $\ell(I_k) \geq 2$. Since $\ell(I_j) = 1$, the results of the execution of I_j are directly observable while those of I_k are not. Hence the behavior of the microprocessor as observed at its external pins under the fault $f(I_j/I_k)$ is the same as it would be under the fault $f(I_j/\phi)$. Therefore Procedures 4.2 and 4.3 given in Section 4.3.2.1 should be followed in this case.

Case B(1.2): $\ell(I_j) = \ell(I_k) = 1$. Many of the faults in this case would be readily detected because I_j and I_k have the highest observability. For example, I_j and I_k may read and write data into the main memory during different machine cycles of the corresponding instruction cycles; or during the execution of I_j and I_k a different sequence of status signals may be emitted on the status pins. Therefore we will explicitly discuss the detection of those faults which cannot be easily detected in this fashion. This case can be further divided into two subcases, namely, case B(1.2.1) and case B(1.2.2) as described below.

Table 4.2. Cases for the details of test generation for detecting $f(I_j/I_k)$

Case and its description				Test generation procedures	Theorems proving the fault coverage
Case B(1) $\ell(I_j) = 1$	Case B(1.1) $\ell(I_k) \geq 2$			Procedures 4.2 and 4.3	*
	Case B(1.2) $\ell(I_k) = 1$	Case B(1.2.1) During the execution of I_j and I_k the results of the operations performed are read out.		Procedure 4.6	Theorem 4.5
		Case B(1.2.2) During the execution of I_j and I_k the results of the operations performed are stored in registers, and not read out.	Case B(1.2.2.1) Results produced by I_j and I_k are stored in the same register.	Procedure 4.7	Theorem 4.6
			Case B(1.2.2.2) Results produced by I_j and I_k are stored in different registers.	Procedure 4.8	Theorem 4.7
Case B(2) $\ell(I_j) = 2$	Case B(2.1) $D(I_j) \neq D(I_k)$			Procedure 4.4	Theorem† 4.2
	Case B(2.2) $D(I_j) = D(I_k)$			Procedure 4.17	Theorem 4.16
Case B(3)	Case B(3.1) $D(I_j) \neq D(I_k)$			Procedures 4.4 and 4.5	Theorems† 4.3 and 4.4
$\ell(I_j) = K \geq 3$	Case B(3.2) $D(I_j) = D(I_k)$	Case B(3.2.1) $\ell(S(I_k)) < K$		Procedure 4.9	Theorem 4.8
		Case B(3.2.2) $\ell(S(I_k)) = K$		Procedure 4.10	Theorem 4.9

*Fault coverage follows immediately from the description of the procedure.
†Fault coverage can be proved by following arguments similar to those in the proof of the theorem (marked with †).

The tests generated by Procedure 4.6 for case B(1.2.1) are to be applied before those generated by Procedures 4.7 and 4.8 for case B(1.2.2).

Case B(1.2.1): During the execution of instructions I_j and I_k the results of the operations performed are "read out". The operation performed could be as simple as a data transfer from a register to the main memory. Examples of this case are given below.

Example 4.6: Instruction I_j is "Store the contents of register R_1 into the main memory using direct addressing," and instruction I_k is "Store the contents of register R_2 into the main memory using direct addressing."

Now consider another example. Instruction I_j is "Add the contents of the accumulator and the contents stored at the top of a LIFO stack (maintained in the main memory) and store the result at the top of the stack," and instruction I_k is "Subtract the contents of the accumulator from the contents stored at the top of the LIFO stack and store the result at the top of the stack."

Procedure 4.6:

This procedure is applicable for case B(1.2.1). It generates tests to detect fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = 1$ and during the execution of I_j and I_k the results of the operations are read out.

Step 1: Store proper operands in $S(I_j)$ and $S(I_k)$ such that when I_j is executed RESULT 1 is read out and when I_k is executed RESULT 2 is read out, and RESULT 1 $\neq$ RESULT 2.

Step 2: Execute I_j. /Expected output data = RESULT 1/

Step 3: Execute I_k. /Expected output data = RESULT 2/

Theorem 4.5: Procedure 4.6 detects $f(I_j/I_k)$ in case B(1.2.1).

Proof: If the proper operands are really stored in $S(I_k)$ in step 1 so that RESULT 2 is read out when I_k is executed, the fault will be detected in step 2 itself, as RESULT 2 will be read out (instead of expected output data = RESULT 1).

On the other hand, if the required operands are not stored in $S(I_k)$ due to faults involved in the instructions used to write data in $S(I_k)$, the fault may not be detected in step 2; the execution of I_k may read out RESULT 1 as the wrong operands are stored in $S(I_k)$. But in this case the fault will be detected in step 3 as the execution of I_k will produce RESULT 1. Note that if $f(I_j/I_k)$ is present we assume that I_k is correctly executed. (Recall the fault model in Section 3.2.) □

Case B(1.2.2): During the execution of instructions I_j and I_k the results of the operations performed are stored in registers. (The results are not read out as in case B(1.2.1).) The operation performed could be as simple as a data transfer from a main memory location to a register. In this situation the instruction belongs to class T. If the operation involves some data manipulation, the instruction belongs to class M.

Note that $\ell(I_j) = \ell(I_k) = 1$. If the instruction I_j or I_k belongs to class M, at least one of the operands for I_j or I_k must be stored in the main memory. (If all the operands for I_j and I_k are available in registers we would have $\ell(I_j)$, $\ell(I_k) \geq 2$. Refer to step 4 of the labeling algorithm given in Section 4.1.) The address of the operand stored in the main memory is computed and then transferred from a register holding

it to the address register of the main memory. Thus the register storing the address of the operand is implicitly read out during the execution of I_j and I_k. This is precisely the reason to assign label 1 to I_j and I_k. (Refer to step 3 of the labeling algorithm.) As mentioned earlier, the results of operations performed under I_j and I_k are stored in registers. If the instruction belongs to class M, the result produced is stored in a register with label 1. This is consistent with the assumption (regarding the label of an instruction of class M) made in Section 4.3.1.

If the instruction belongs to class T the data transferred from the main memory may be stored in a register with label greater than 1. In this situation fault detection may or may not be easy as illustrated in the following example.

Example 4.7: Let instruction I_j be "Load the contents of the memory location pointed to by register R_1 in register R_2," and let instruction I_k be "Load the contents of the memory location pointed to by register R_3 in register R_2." Both instructions store the result of their operation (which is a simple data transfer) in register R_2. Even if $\ell(R_2) \geq 2$, fault $f(I_j/I_k)$ can be easily detected by choosing different addresses (pointers) in R_1 and R_3. If $f(I_j/I_k)$ is present, the address stored in R_3 will be (implicitly) read out on the address bus instead of the address stored in R_1 when I_j is executed, and the fault will be detected. Thus this case is really not different from case B(1.2.1).

We now consider another example where the fault detection is not so easy. Let instruction I_j be "Load the contents of the memory location pointed to by register R_1 in register R_2," and let instruction I_k be "Load the contents of the memory location pointed to by register R_1 in

register R_3." In this case $f(I_j/I_k)$ can be detected by storing different data in R_2 and R_3 and then reading out these registers by executing READ (R_2) and READ (R_3). If $\ell(R_2)$ or $\ell(R_3)$ is greater than 1, (i.e., READ (R_2) or READ (R_3) contain instructions with label greater than 1) it is not guaranteed that READ (R_2) and READ (R_3) sequences can correctly read out R_2 and R_3 because the microprocessor under test has not yet executed tests to detect $f(I_p/I_q)$, $f(I_p/I_p+I_q)$ where $\ell(I_p)$, $\ell(I_q) \geq 2$. In this case we treat I_j and I_k as if they have label $\ell(R_2) + 1$ and $\ell(R_3) + 1$, respectively, as far as the test generation for faults $f(I_j/I_k)$, $f(I_j/I_j+I_k)$, $f(I_k/I_j)$, or $f(I_k/I_k+I_j)$ is concerned. □

Case B(1.2.2) being considered applies only to those instructions which store their result in a register with label 1. Depending on whether the results of I_j and I_k are stored in the same or different registers we divide this case further into two subcases.

Case B(1.2.2.1): The results produced by executing instructions I_j and I_k are stored in the same register. Let this register be designated as R_p; furthermore, let READ $(R_p) = \langle I_p \rangle$. Note that $\ell(I_p) = 1$. An example of this case is given below.

Example 4.8: Instruction I_j is "Add the contents of the accumulator and the contents of the memory location (next to the one storing the opcode of instruction I_j) and store the result in the accumulator," and instruction I_k is "Subtract the contents of the accumulator from the contents of the memory location and store the result in the accumulator." □

Procedure 4.7:

This procedure is applicable for case B(1.2.2.1). It generates tests to detect fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = 1$, and during

the execution of I_j and I_k the results of the operations are stored in the same register R_p.

Step 1: Store proper operands in $S(I_j)$ and $S(I_k)$ such that when I_j is executed RESULT 1 is produced and when I_k is executed RESULT 2 is produced, and RESULT 1 $\neq$ RESULT 2

Step 2: If $R_p \in S(I_j) \cup S(I_k)$ then read out register R_p by executing READ $(R_p) = \langle I_p \rangle$. /To make sure that R_p contains proper operand to be stored in step 1. The other member of $S(I_j)$ or $S(I_k)$ is a location in the main memory./

Step 3: If $R_p \in S(I_j) \cup S(I_k)$ then repeat step 2.

Step 4: Execute I_j.

Step 5: Read out register R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = RESULT 1 /

□

Theorem 4.6: Procedure 4.7 detects $f(I_j/I_k)$ in case B(1.2.2.1).

Proof: If the register R_p is a member of $S(I_j)$ or $S(I_k)$ it must be ensured that it contains the proper operand to be stored in step 1, otherwise I_k could produce RESULT 1 instead of RESULT 2 and fault masking would occur.

The instruction I_p involved in step 2 will correctly read out R_p because $\ell(I_p) = 1$, and the microprocessor under test would have already executed the tests to detect $f(I_p/I_q)$ where $\ell(I_q) = 1$ (generated by Procedure 4.6 for case B(1.2.1)); however, if the fault $f(I_p/I_p+I_{p'})$ is present, where $\ell(I_{p'}) = 2$ and $D(I_{p'}) = \{R_p\}$, the contents of R_p may change after it is read out by executing I_p. If this happens it will be detected in step 3. On the other hand if the contents of R_p do not change

after the first execution of I_p (in step 2), they will not change after the second execution of I_p (in step 3) either.

In step 4 I_j is executed. If $f(I_j/I_k)$ is present, RESULT 2 will be produced in R_p in step 4, and it will be detected when R_p is read out in step 5. □

Case B(1.2.2.2): The results produced by executing instructions I_j and I_k are stored in different registers. Let the result produced by I_j be stored in register R_p. Furthermore, let READ $(R_p) = \langle I_p \rangle$, and $\ell(I_p) = 1$. An example of this case is given below.

Example 4.9: Instruction I_j is "Load register R_1 from the main memory location (next to the one storing the opcode of instruction I_j)," and instruction I_k is "Load register R_2 from the main memory location." □

Procedure 4.8:

This procedure is applicable for case B(1.2.2.2). It generates tests to detect fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = 1$, and during the execution of I_j and I_k the results of the operations are stored in different registers.

Step 1: Store OPERAND 1 in register R_p and proper operands in $S(I_j)$ such that when I_j is executed RESULT 1 is produced in R_p, and RESULT 1 $\neq$ OPERAND 1.

Step 2: Read out register R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = OPERAND 1/

Step 3: Repeat step 2.

Step 4: Execute I_j.

Step 5: Read out register R_p by executing READ $(R_p) = \langle I_p \rangle$.

/Expected output data = RESULT 1 ≠ OPERAND 1/ □

Theorem 4.7: Procedure 4.8 detects $f(I_j/I_k)$ in case B(1.2.2.2).

The proof of this theorem follows the proof of Theorem 4.6 closely and will not be repeated here.

4.3.3.2. Test Generation for $f(I_j/I_k)$ when $\ell(I_j) = 2$

This case is referred to as case B(2) and is divided into two subcases depending on whether or not $D(I_j) = D(I_k)$.

Case B(2.1): $D(I_j) \neq D(I_k)$. In this case under the fault $f(I_j/I_k)$, the contents of $D(I_j)$ remain unchanged as they would remain under the fault $f(I_j/\phi)$. Hence, the procedure for this case is the same as Procedure 4.4. Furthermore, using arguments similar to those in the proof of Theorem 4.2, it can be proved that Procedure 4.4 detects $f(I_j/I_k)$ in this case.

Case B(2.2): $D(I_j) = D(I_k)$. This case is not treated here because all the details considered for case C(3.1) in Section 4.3.4.3 apply to this case. Procedure 4.17 used to detect $f(I_j/I_j+I_k)$ in case C(3.1) (refer to Table 4.3) can detect $f(I_j/I_j)$ in case B(2.2) discussed here. This will be pointed out in the proof of Theorem 4.16.

4.3.3.3. Test Generation for $f(I_j/I_k)$ When $\ell(I_j) = K \geq 3$

This case is referred to as case B(3). Note that I_j belongs to class T according to our assumption in Section 4.3.1. This case can be further divided into two subcases depending on whether or not $D(I_j) = D(I_k)$.

Case B(3.1): $D(I_j) \neq D(I_k)$. In this case under the fault $f(I_j/I_k)$, the contents of $D(I_j)$ remain unchanged as under the fault $f(I_j/\phi)$. Hence, the procedures for this case will be the same as Procedures 4.4 and 4.5. Note that Procedure 4.4 is applied in case A(3.1) and Procedure 4.5 is applied in case A(3.2). Using arguments similar to those in the proofs of Theorems 4.3 and 4.4, it can be proved that Procedures 4.4 and 4.5 detect $f(I_j/I_k)$ in this case.

Case B(3.2): $D(I_j) = D(I_k)$. Note that $\ell(I_j) = \ell(I_k) = K$, and $\ell(D(I_j)) = \ell(D(I_k)) = K-1$. Therefore no instruction in the READ $(D(I_j))$ sequence can have a label greater than K-1. This case is illustrated in Figure 4.4. Note that READ $(D(I_j)) = \langle I_{p_{K-1}}, I_{p_{K-2}}, \ldots, I_{p_1} \rangle$ where $\ell(I_{p_i}) = i$, for $1 \leq i \leq K-1$. Depending on $\ell(S(I_k))$ the case can be further divided into two subcases. Case B(3.2.1) applies when $\ell(S(I_k)) < K$, and case B(3.2.2) applies when $\ell(S(I_k)) = K$. Note that $\ell(S(I_k)) \not> K$, since $\ell(I_k) = K$ and I_k belongs to class T. (Also refer to the labeling algorithm given in Section 4.1.)

Case B(3.2.1): $\ell(S(I_k)) < K$. In this case the instruction I_k does not belong to the READ $(S(I_k))$ sequence. (This will be proved in the proof of Theorem 4.8.)

Procedure 4.9:

This procedure is applicable for case B(3.2.1). It generates tests to detect fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = K \geq 3$, $D(I_j) = D(I_k)$, and $\ell(S(I_k)) < K$.

Step 1: Store OPERAND 1 in $S(I_j)$ and OPERAND 2 in $S(I_k)$ such that OPERAND 1 $\neq$ OPERAND 2.

Step 2: Execute I_j.

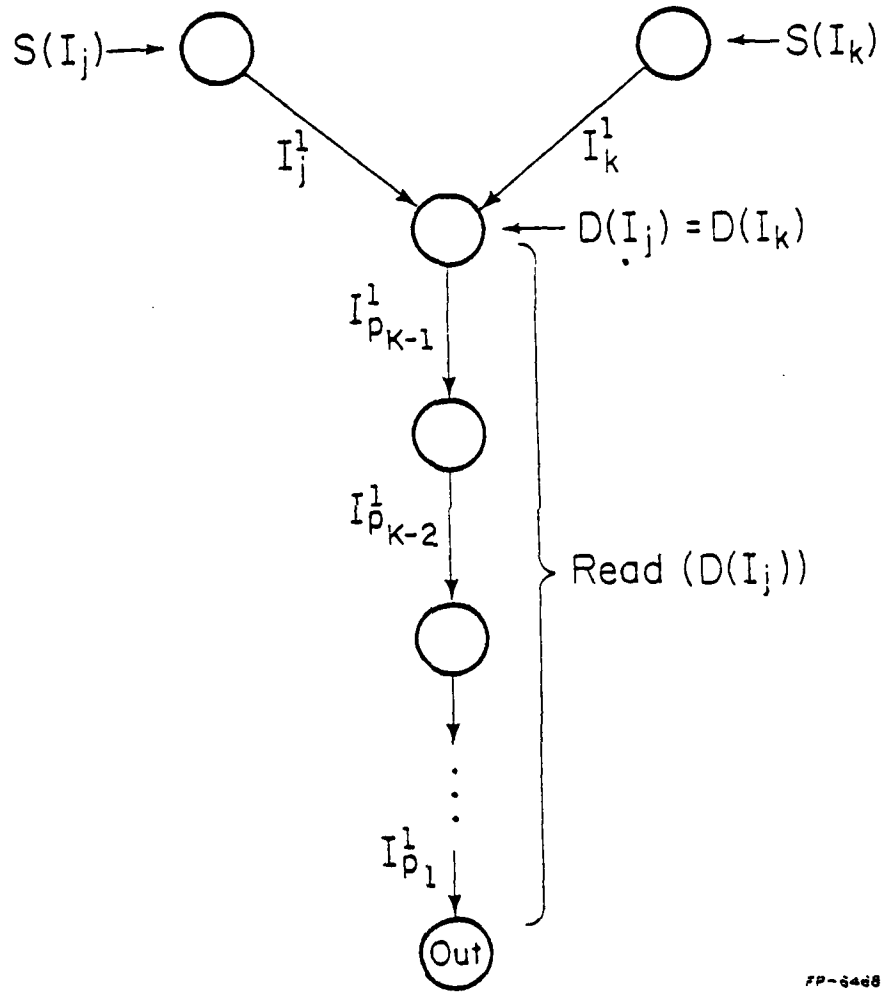


Figure 4.4. Illustrating case B(3.2) in Section 4.3.3.3.

Step 3: Read out $S(I_k)$ by executing READ $(S(I_k))$.

/Expected output data = OPERAND 2/

Step 4: Read out $D(I_j)$ by executing READ $(D(I_j))$.

/Expected output data = OPERAND 1/ □

Theorem 4.8: Procedure 4.9 detects $f(I_j/I_k)$ in case B(3.2.1).

Proof: Note that $\ell(I_k) = K$. Since $\ell(S(I_k)) < K$, no instruction in the READ $(S(I_k))$ sequence can have a label greater than K-1, implying that $I_k \notin$ READ $(S(I_k))$. Therefore READ $(S(I_k))$ reads out $S(I_k)$ without routing it through $D(I_j) = D(I_k)$.

When the microprocessor under test executes this procedure, it has already executed the tests required to detect $f(I_p/\phi)$, $f(I_p/I_q)$, and $f(I_p/I_p+I_q)$ where $1 \leq \ell(I_p)$, $\ell(I_q) \leq K-1$, and $f(I_v/I_v+I_w)$ where $1 \leq \ell(I_v) \leq K-1$ and $\ell(I_w) = K$. Therefore, when the microprocessor under test executes this procedure it has already been checked that the READ $(D(I_j))$ and READ $(S(I_k))$ sequences correctly read out $D(I_j)$ and $S(I_k)$, respectively. (Recall that no instruction in the READ $(D(I_j))$ or READ $(S(I_k))$ sequences can have a label greater than K-1.) Moreover, after the execution of these sequences the contents of $D(I_j)$ and $S(I_k)$ do not alter.

In step 2, I_j is executed and it is expected to produce OPERAND 1 in $D(I_j)$. If $f(I_j/I_k)$ is present, OPERAND 2 will be produced in $D(I_j)$ instead of OPERAND 1, provided $S(I_k)$ really contained OPERAND 2 when I_j is executed. In this situation the fault will be detected in step 4. (Note that in step 3, READ $(S(I_k))$ reads out $S(I_k)$ without routing it through $D(I_j) = D(I_k)$.) On the other hand, due to faults involved in the instructions used to write data in $S(I_k)$, OPERAND 2 may not be stored

in $S(I_k)$; in this case the fault will be detected in step 3 itself. □

Case B(3.2.2): $\ell(S(I_k)) = K$. In this case $I_k \in$ READ $(S(I_k))$. Procedure 4.9 used for case B(3.2.1) cannot be used in this case because in step 3, READ $(S(I_k))$ will read out $S(I_k)$ by routing it through $D(I_j) = D(I_k)$, destroying its contents.

Procedure 4.10:

This procedure is applicable for case B(3.2.2). It generates tests to detect fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = K \geq 3$, $D(I_j) = D(I_k)$, and $\ell(S(I_k)) = K$.

Step 1: Store OPERAND 1 in $S(I_j)$ and OPERAND 2 in $S(I_k)$ such that OPERAND 1 ≠ OPERAND 2.

Step 2: For i ← 1 TO K DO

BEGIN

Execute I_j;

Read out $D(I_j)$ by executing READ $(D(I_j))$.

/Expected output data = OPERAND 1/

END

Step 3: Execute I_k.

Step 4: Read out $D(I_k)$ by executing READ $(D(I_k))$ = READ $(D(I_j))$.

/Expected output data = OPERAND 2/ □

Theorem 4.9: Procedure 4.10 detects $f(I_j/I_k)$ in case B(3.2.2).

Proof: In step 2 of Procedure 4.10, register $D(I_j)$ is read out by executing READ $(D(I_j)) = \langle I_{p_{K-1}}, I_{p_{K-2}}, \ldots, I_{p_1} \rangle$. (Refer to Figure 4.4.) Recall that $\ell(I_j) = \ell(I_k) = K$, and $\ell(I_{p_i}) = i$, for $1 \leq i \leq K-1$. When the microprocessor under test executes this procedure,

it has already been checked that $f(I_p/\phi)$, $f(I_p/I_q)$ and $f(I_p/I_p+I_q)$ do not exist, where $1 \leq \ell(I_p), \ell(I_q) \leq K-1$. Therefore, $D(I_j)$ can be correctly read out by executing READ $(D(I_j))$.

If OPERAND 2 is really stored in $S(I_k)$ in step 1, the fault $f(I_j/I_k)$ will be detected when $D(I_j)$ is read out during the first iteration of step 2, since OPERAND 2 will be read out instead of OPERAND 1. On the other hand, due to the faults involved in instructions used to write data in $S(I_k)$, OPERAND 1 may have been stored in $S(I_k)$. In this situation the fault will not be detected during the first iteration of step 2. We now prove that it is necessary and sufficient to repeat the loop in step 2 K times, where $|\text{READ } (D(I_j))| = K-1$.

Sufficiency: So far the microprocessor under test has not executed the tests to detect $f(I_v/I_v+I_w)$, where $1 \leq \ell(I_v) \leq K-1$ and $K+1 \leq \ell(I_w) \leq K_{max}$. If a fault $f(I_{p_i}/I_{p_i} + I_{p_i'})$ is present, where $I_{p_i} \in$ READ $(D(I_j))$ and $K+1 \leq \ell(I_{p_i'}) \leq K_{max}$, it has yet not been detected. Consider a sequence of instructions $\langle I_{p_{K-1}'}, I_{p_{K-2}'}, \ldots, I_{p_1'} \rangle$ where $I_{p_i'}$ belongs to this sequence if and only if the fault $f(I_{p_i}/I_{p_i} + I_{p_i'})$ is present, for $1 \leq i \leq K-1$. Since $K+1 \leq \ell(I_{p_i'}) \leq K_{max}$ and $K \geq 3$, $I_{p_i'}$ belongs to class T according to our assumption in Section 4.3.1. Therefore, we designate the sequence $\langle I_{p_{K-1}'}, I_{p_{K-2}'}, \ldots, I_{p_1'} \rangle$ as the T sequence σ'. Note that there are at most K-1 instructions in σ'. Thus when the READ $(D(I_j))$ sequence is executed in step 2, the T sequence σ' is also executed in addition. Consider the register which is 1-step transferrable to $S(I_k)$ under the T sequence σ'. The contents of this register before the execution of READ $(D(I_j))$ in the first iteration of step 2 will become the final contents of $S(I_k)$ at the end of the first

iteration of step 2. This may alter the contents of $S(I_k)$.

During the first K-1 iterations of step 2 READ $(D(I_j))$ is executed K-1 times, and so is the T sequence σ'. If register $S(I_k)$ contains OPERAND 1 at the end of each of i iterations of step 2, (i.e., at the end of each of i executions of the T sequence σ', for $1 \leq i \leq K-1$), then by Corollary 2.3, at the end of the K^{th} execution of the T sequence σ', $S(I_k)$ will contain OPERAND 1. Recall that there are at most K-1 instructions in the T sequence σ'. If this is so, the fault will be detected in step 4 as OPERAND 1 $\neq$ OPERAND 2 will be read out.

On the other hand, if at the end of any of the first K-1 iterations of step 2, $S(I_k)$ contains data different from OPERAND 1, the fault will be detected in the next iteration of step 2 when $D(I_j)$ is read out by executing READ $(D(I_j))$, as the output data will be different from OPERAND 1.

Necessity: Let the loop in step 2 be repeated only p times, where $1 \leq p \leq K-1$. (Recall that $K \geq 3$.) Consider the register which is p-step transferrable to $S(I_k)$ under the T sequence σ'. Since there can be as many as K-1 instructions in σ', such a register can exist (from Theorem 2.1). If this register contains OPERAND 2, and $S(I_k)$ and each register which is j-step transferrable to $S(I_k)$ $(1 \leq j \leq p-1)$ under the T sequence σ' contains OPERAND 1 at the beginning of the first iteration of the loop, then the contents of $S(I_k)$ will be equal to OPERAND 1 at the end of each of i iterations of the loop, for $1 \leq i \leq p-1$, and they will become OPERAND 2 at the end of the p^{th} iteration of the loop. (Refer to Corollary 2.1.)

Therefore, OPERAND 1 will be read out during each of p iterations of the loop in step 2, and OPERAND 2 will be read out in step 4, as expected. Thus the fault may remain undetected if the loop in step 2 is repeated less than K times. □

4.3.4. Test Generation for $f(I_j/I_j+I_k)$

The details of test generation depend on $\ell(I_j)$ and $\ell(I_k)$. We consider six cases, namely, case C(1) through case C(6). The suffix C is used to denote that a case belongs to the details of test generation for $f(I_j/I_j+I_k)$. These cases are divided into subcases which are listed in Table 4.3. For each case, the table gives which test generation procedure is applicable and which theorem proves the fault coverage.

4.3.4.1. Test Generation for $f(I_j/I_j+I_k)$ When $\ell(I_j) = \ell(I_k) = 1$

This case is referred to as case C(1). As in the case B(1.2), many faults in this case would be readily detected because I_j and I_k have the highest observability. This case can be divided into two subcases, namely case C(1.1) and case C(1.2), exactly in the same way case B(1.2) was divided into case B(1.2.1) and case B(1.2.2). (Refer to Section 4.3.3.1.) The tests generated by Procedure 4.11 for case C(1.1) are to be applied before those generated by procedures for case C(1.2).

Case C(1.1): During the execution of I_j and I_k the results of the operation performed are "read out".

Procedure 4.11:

This procedure is applicable for case C(1.1). It generates tests to detect $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = 1$ and during the execution of I_j and I_k the results of the operations are read out.

Table 4.3. Cases for the details of test generation for detecting $f(I_j/I_j+I_k)$.

Case and its description			Test generation Procedures	Theorems proving the fault coverage
Case C(1)	Case C(1.1) During the execution I_j and I_k the results of the operations performed are read out.		Procedure 4.11	Theorem 4.10
$\ell(I_j)=\ell(I_k)=1$	Case C(1.2) During the execution of I_j and I_k the results of the operations performed are stored in registers, and not read out.	Case C(1.2.1) Results produced by I_j and I_k are stored in the same register.	Procedure 4.7 (with two modifications)	Theorems† 4.6 and 4.10
		Case C(1.2.2) Results produced by I_j and I_k are stored in different registers.	Procedure 4.12	Theorem 4.11
Case C(2)	Case C(2.1) $D(I_k) \subseteq S(I_j)$, and the result of the operation performed under I_j is read out during its execution.		Procedure 4.13	Theorem 4.12
$\ell(I_j) = 1$, $\ell(I_k) = 2$	Case C(2.2) $D(I_k) \not\subseteq S(I_j)$, and the result of the operation performed under I_j is read out during its execution.		Procedure 4.14	Theorem 4.13
	Case C(2.3) Results produced by I_j and I_k are stored in the same register.		Procedure 4.15	Theorem 4.14
	Case C(2.4) Results produced by I_j and I_k are stored in different registers.		Procedure 4.16	Theorem 4.15
Case C(3)	Case C(3.1) $D(I_j) = D(I_k)$		Procedure 4.17	Theorem 4.16
$\ell(I_j)=\ell(I_k)=2$	Case C(3.2) $D(I_j) \neq D(I_k)$		Procedure 4.18	Theorem 4.17

†Fault coverage can be proved by following arguments similar to those in the proof of the theorem (marked with †).

Table 4.3 (continued). Cases for the details of test generation for detecting $f(I_j/I_j+I_k)$.

Case and its description			Test generation procedure	Theorems proving the fault coverage
Case C(4) $\ell(I_j) = \ell(I_k) = K \geq 3$	Case C(4.1) $D(I_j) = D(I_k)$	Case C(4.1.1) $\ell(S(I_k)) < K$	Procedure 4.19	Theorem 4.18
		Case C(4.1.2) $\ell(S(I_k)) = K$	Procedure 4.20	Theorem 4.19
	Case C(4.2) $D(I_j) \neq D(I_k)$	Case C(4.2.1) $\ell(S(I_k)) < K$	Procedure 4.21	Theorem 4.20
		Case C(4.2.2) $\ell(S(I_k)) = K$	Procedure 4.22	Theorem 4.21
Case C(5) $1 \leq \ell(I_j) \leq K$, $\ell(I_k) = K+1$, and $K \geq 2$	Case C(5.1) $\ell(S(I_k)) \leq K-1$		Procedure 4.21	Theorem† 4.20
	Case C(5.2) $\ell(S(I_k)) = K$, or $K+1$		Procedure 4.22	Theorem† 4.9
Case C(6) $K+1 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = K$	Case C(6.1) $2 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = 1$		Readily detected by executing I_j	-
	Case C(6.2) $3 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = 2$		Procedure 4.23	Theorems† 4.10 through 4.17
	Case C(6.3) $K+1 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = K \geq 3$		Procedure 4.24	Theorems† 4.10 through 4.17

†Fault coverage can be proved by following arguments similar to those in the proof of the theorem (marked with †).

Step 1: If possible, store proper operands in $S(I_j)$ and $S(I_k)$ such that when I_j is executed RESULT 1 is read out and when I_k is executed RESULT 2 is read out, and (RESULT 1) $\vee$ (RESULT 2) $\neq$ (RESULT 1). /$\vee$ denotes the bit-wise logical OR function/

Step 2: Execute I_j. /Expected output data = RESULT 1 /

Step 3: Execute I_k. /Expected output data = RESULT 2 /

Step 4: If possible, store proper operands in $S(I_j)$ and $S(I_k)$ such that when I_j is executed RESULT 3 is read out and when I_k is executed RESULT 4 is read out, and (RESULT 3) $\wedge$ (RESULT 4) $\neq$ (RESULT 3). /$\wedge$ denotes the bit-wise logical AND function/

Step 5: Execute I_j. /Expected output data = RESULT 3/

Step 6: Execute I_k. /Expected output data = RESULT 4/ □

The underlined phrase "if possible" in steps 1 and 4 may come as a surprise. However, it may not be possible to satisfy conditions given in both the steps due to the nature of the operations performed under the instructions I_j and I_k. The following example illustrates the point.

Example 4.10: Let instruction I_j be "Store the contents of register R_1 into the main memory location (next to the one containing the opcode of instruction I_j)," and instruction I_k be "Perform the logical AND of the contents of registers R_1 and R_2 and store the result into the main memory location." The requirement in step 4 of Procedure 4.11 can be satisfied by storing a ONE in R_1 and a ZERO in R_2 so that under I_j a ONE is read out and under I_k a ZERO is read out, and (ONE $\wedge$ ZERO) = ZERO $\neq$ ONE.

On the other hand, no operands would satisfy the requirement in step 1. This can be easily proved. Let OPERAND 1 and OPERAND 2 be stored in R_1 and R_2 respectively. I_j will read out RESULT 1 = OPERAND 1 and I_k will read out RESULT 2 = (OPERAND 1) $\wedge$ (OPERAND 2). If $f(I_j/I_j+I_k)$ is present, then as illustrated in Example 3.2, (RESULT 1) * (RESULT 2) would be read out, where * denotes the bit-wise logical AND or OR function depending on technology. In this situation, if * is the OR function, I_j would read out (RESULT 1) $\vee$ (RESULT 2) = (OPERAND 1) $\vee$ ((OPERAND 1) $\wedge$ (OPERAND 2)) = OPERAND 1 = RESULT 1, as expected. Thus, the fault $f(I_j/I_j+I_k)$ is an undetectable fault. □

Theorem 4.10: Procedure 4.11 detects all detectable $f(I_j/I_j+I_k)$ faults in case C(1.1).

Proof: If the proper operands as required by the condition in step 1 are really stored in $S(I_k)$, and $f(I_j/I_j+I_k)$ causes the actual result read out under instruction I_j to be the bit-wise logical OR combination of RESULT 1 and RESULT 2, the fault will be detected in step 2 as (RESULT 1) $\vee$ (RESULT 2) $\neq$ RESULT 1 will be read out. On the other hand, under faults involved in the instructions used to store data in $S(I_k)$, wrong operands may have been stored in $S(I_k)$ in step 1. In this situation the fault will be detected not in step 2 but in step 3 when I_k is executed. (Recall the fault model in Section 3.2; if $f(I_j/I_j+I_k)$ exists, I_k is correctly executed.) Steps 4, 5, and 6 detect the fault if $f(I_j/I_j+I_k)$ causes the actual result read out under instruction I_j to be the bit-wise logical AND combination of RESULT 1 and RESULT 2. □

Case C(1.2): During the execution of instructions I_j and I_k the results of the operations performed are stored in registers. (The results are not read out as in case C(1.1).) This case is identical to case B(1.2.2), and all the points illustrated in Example 4.7 do apply here too. Depending on whether the results of I_j and I_k are stored in the same or different registers we divide this case into two subcases, namely, case C(1.2.1) and case C(1.2.2), exactly in the same way case B(1.2.1) was divided into case B(1.2.1.1) and case B(1.2.1.2).

Case C(1.2.1): The results produced by executing instructions I_j and I_k are stored in the same register. Let this register be designated as R_p; furthermore let READ $(R_p) = \langle I_p \rangle$. Note that $\ell(I_p) = 1$. (Refer to Example 4.8.) In this case we follow Procedure 4.7 given for case B(1.2.2.1), except for two modifications:

1. In step 1 of Procedure 4.7, RESULT 1 and RESULT 2 should satisfy the condition (RESULT 1) $\vee$ (RESULT 2) $\neq$ RESULT 1.

2. Step 1 through step 5 of Procedure 4.7 are repeated, and RESULT 1 and RESULT 2 should satisfy the condition (RESULT 1) $\wedge$ (RESULT 2) $\neq$ RESULT 1.

It can be easily proved that the procedure detects $f(I_j/I_j+I_k)$ in this case by following the similar arguments used in the proofs of Theorems 4.6 and 4.10.

Case C(1.2.2): The results produced by executing instructions I_j and I_k are stored in different registers. Let the result produced by I_k be stored in register R_p. Furthermore, let READ $(R_p) = \langle I_p \rangle$. Note that $\ell(I_p) = 1$.

Procedure 4.12:

This procedure is applicable for case C(1.2.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = 1$, and during the execution of I_j and I_k the results of the operations are stored in different registers.

Step 1: Store OPERAND 1 in register R_p and proper operands in $S(I_k)$ such that when I_k is executed RESULT 1 is stored in R_p and RESULT 1 $\neq$ OPERAND 1.

Step 2: Read out register R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = OPERAND 1 /

Step 3: Repeat step 2.

Step 4: Execute I_j.

Step 5: Read out register R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = OPERAND 1 $\neq$ RESULT 1/ □

Theorem 4.11: Procedure 4.12 detects $f(I_j/I_j+I_k)$ in case C(1.2.2).

The proof of this theorem follows closely that of Theorem 4.6 and will not be repeated here. □

4.3.4.2. Test Generation for $f(I_j/I_j+I_k)$ When $\ell(I_j) = 1$ and $\ell(I_k) = 2$

This case is referred to as case C(2). We divide this case into various subcases, namely, case C(2.1) through case C(2.4), depending on whether $D(I_k) \subseteq S(I_j)$ and whether during the execution of I_j the result of the operation performed under I_j is read out. Tests generated for case C(2.1) and case C(2.2) are to be applied before those for case C(2.3) and case C(2.4).

Case C(2.1): $D(I_k) \subseteq S(I_j)$, and the result of the operation performed under I_j is read out during its execution. An example of this case is given below.

Example 4.11: Instruction I_j is "Push the contents of the accumulator on the top of a LIFO stack maintained in the main memory," and instruction I_k is "Transfer the contents of register R_1 to the accumulator." In general, I_j may perform some operation on its operands and then read out the result. □

Procedure 4.13:

This procedure is applicable for case C(2.1). It generates tests to detect $f(I_j/I_j+I_k)$ when $\ell(I_j) = 1$, $\ell(I_k) = 2$, $D(I_k) \subseteq S(I_j)$, and the result of the operation performed under I_j is read out during its execution.

Step 1: Store proper operands in $S(I_j)$ such that when I_j is executed RESULT 1 is read out. Store proper operands in $S(I_k)$ such that when I_k is executed, it changes the contents of $D(I_k)$ so that if I_j is executed after I_k, RESULT 2 ≠ RESULT 1 will be read out.

Step 2: Execute I_j. /Expected output data = RESULT 1 /

Step 3: Repeat step 2. /Expected output data = RESULT 1 ≠ RESULT 2 /

Step 4: Execute I_k.

Step 5: Execute I_j. /Expected output data = RESULT 2 / □

Theorem 4.12: Procedure 4.13 detects $f(I_j/I_j+I_k)$ in case C(2.1).

Proof: If the proper operands as required in step 1 are really stored in $S(I_k)$ the fault will be detected in step 3, because RESULT 2 will be read out instead of RESULT 1. If proper operands are not stored in $S(I_k)$ due to the faults in instructions used to write data in $S(I_k)$, the

fault may not be detected in step 3, but it will be detected in step 5 because the output data will then be different from RESULT 2. □

Case C(2.2): $D(I_k) \not\subseteq S(I_j)$ and the result of the operation performed under I_j is read out during its execution.

Procedure 4.14:

This procedure is applicable for case C(2.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = 1$, $\ell(I_k) = 2$, $D(I_k) \not\subseteq S(I_j)$, and the result of the operation performed under I_j is read out during its execution.

Step 1: Store proper operands in $S(I_k)$ and OPERAND 1 in $D(I_k)$ such that when I_k is executed it produces RESULT 1 $\neq$ OPERAND 1 in $D(I_k)$.

Step 2: Execute I_j.

Step 3: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = OPERAND 1/

Step 4: If $\ell(S(I_k)) \geq 2$ then repeat steps 2 and 3 else go to step 5.

Step 5: Execute I_k.

Step 6: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = RESULT 1 $\neq$ OPERAND 1/ □

Theorem 4.13: Procedure 4.14 detects $f(I_j/I_j+I_k)$ in case C(2.2).

Proof: If the proper operands as required in step 1 are really stored in $S(I_k)$ the fault $f(I_j/I_j+I_k)$ will be detected in step 3 because RESULT 1 $\neq$ OPERAND 1 will be read out. On the other hand, if proper operands are not stored in $S(I_k)$ because of faults in the instructions used to store data in $S(I_k)$, OPERAND 1 may be produced in $D(I_k)$ when I_k is executed. In this situation step 3 will not detect the fault $f(I_j/I_j+I_k)$. The necessity for step 4 is now explained.

At this stage the microprocessor under test has not executed the tests to detect $f(I_p/I_p+I_q)$ where $\ell(I_p) = 1$ and $\ell(I_q) \geq 3$. If $f(I_p/I_p+I_q)$ is present and $D(I_q) \subseteq S(I_k)$ (requiring $\ell(S(I_k)) \geq 2$), then the execution of READ $(D(I_k))$ in step 3 may cause a change in the contents of $S(I_k)$ due to the additional execution of I_q. This will be detected in step 4 as the output data will be different from OPERAND 1. On the other hand, if the contents of $S(I_k)$ remain unchanged after step 3 they will remain unchanged after step 4 also. In this situation the fault will be detected in step 6 as the output data will be different from RESULT 1. □

Case C(2.3): The results produced during the execution of I_j and I_k are stored in the same register. (The result of the operation performed under I_j is not read out, as in case C(2.1) or case C(2.2).) Let this register be designated as R_p; furthermore let READ $(R_p) = \langle I_p \rangle$. Note that $\ell(I_p) = 1$. (This is consistent with our discussion in Example 4.7.)

Procedure 4.15:

This procedure is applicable for case C(2.3). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = 1$, $\ell(I_k) = 2$, and the results produced during the execution of I_j and I_k are stored in the same register designated as R_p.

Step 1: If possible, store proper operands in $S(I_j)$ and $S(I_k)$ so that when I_j is executed RESULT 1 is produced in R_p and when I_k is executed RESULT 2 is produced in R_p, and (RESULT 1) $\vee$ (RESULT 2) $\neq$ (RESULT 1).

Step 2: Execute I_j.

Step 3: Read out R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = RESULT 1 /

Step 4: If $\ell(S(I_k)) \geq 2$ then repeat steps 2 and 3 else go to step 5.

Step 5: Execute I_k.

Step 6: Read out R_p by executing READ $(R_p) = \langle I_p \rangle$.
/Expected output data = RESULT 2 /

Step 7: Repeat step 1 with the change that (RESULT 1) $\wedge$ (RESULT 2) $\neq$ (RESULT 1).

Step 8: Repeat steps 2 through 6. □

Theorem 4.14: Procedure 4.15 detects $f(I_j/I_j+I_k)$ in case C(2.3). □

The proof of this theorem follows closely those of Theorems 4.10 and 4.13, and hence is not given.

Case C(2.4): The results produced during the execution of I_j and I_k are stored in different registers.

Procedures 4.16:

This procedure is applicable for case C(2.4). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = 1$, $\ell(I_k) = 2$, and the results produced during the execution of I_j and I_k are stored in different registers.

Step 1: If possible, store proper operands in $S(I_k)$ and OPERAND 1 in $D(I_k)$ so that when I_k is executed RESULT 1 is produced in $D(I_k)$ and RESULT 1 $\neq$ OPERAND 1.

Step 2: Execute I_j.

Step 3: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = OPERAND 1/

Step 4: If $\ell(S(I_k)) \geq 2$ then repeat steps 2 and 3 else go to step 5.

Step 5: Execute I_k.

Step 6: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = RESULT 1/ □

Theorem 4.15: Procedure 4.16 detects $f(I_j/I_j+I_k)$ in case C(2.4). □

The proof of this theorem follows the arguments given in the proof of Theorem 4.13, and hence is not given.

4.3.4.3. Test Generation for $f(I_j/I_j+I_k)$ When $\ell(I_j) = \ell(I_k) = 2$

This case is referred to as case C(3) and is divided into two subcases, namely, case C(3.1) and case C(3.2) depending on whether or not $D(I_j) = D(I_k)$.

Case C(3.1): $D(I_j) = D(I_k)$. The basic requirement of test generation in this case is to store proper operands in $S(I_j)$ and $S(I_k)$ such that if I_j is executed it produces an $x \in \{0,1\}$ in some bit (say the p^{th} bit) of $D(I_j)$, and if I_k is executed it produces $\bar{x}$ in the same bit of $D(I_j)$. When I_j is executed $x * \bar{x}$ will be produced in the p^{th} bit of $D(I_j)$, if $f(I_j/I_j+I_k)$ is present. (* denotes the logic AND or OR function.) Note that $x * \bar{x} \neq x$, for $x = 1$ and $* =$ AND, and for $x = 0$ and $* =$ OR. Thus, if $D(I_j)$ is read out after executing I_j the fault will be detected.

If $D(I_j) \neq S(I_j)$ and $D(I_j) \neq S(I_k)$, no specific data need to be stored in $D(I_j)$ in order to satisfy this requirement. On the other hand if $D(I_j) = S(I_j)$ or $D(I_j) = S(I_k)$, in order to produce an x in the p^{th}

bit of $D(I_j)$ when I_j is executed, and to produce $\bar{x}$ in the p^{th} bit $D(I_j)$ when I_k is executed, it may be required that some specific logic value must be stored in the p^{th} bit of $D(I_j)$. If x must be stored in the p^{th} bit of $D(I_j)$, the situation is referred to as "situation A"; if $\bar{x}$ must be stored in the p^{th} bit of $D(I_j)$, the situation is referred to as "situation B." The following example should clarify these two situations.

Example 4.12: Suppose instruction I_j is "Transfer the contents of register R_1 to register R_2," and instruction I_k is "Perform the logical AND operation on the contents of register R_2 and R_3, and store the result in R_2." Instructions I_j and I_k are represented in a graph in Figure 4.5. We must store $x = 0$ in the p^{th} bit of R_1, and $\bar{x} = 1$ in the p^{th} bit of both R_2 and R_3, so that if I_j is executed $x = 0$ is produced in the p^{th} bit of R_2, and if I_k is executed $\bar{x} = 1$ is produced in the p^{th} bit of R_2. Thus, this is an example of what we have referred to as situation B. An example of situation A can be easily obtained by simply renaming instruction I_j as I_k, and vice-versa, and letting $x = 1$. □

We now present two subprocedures to be used in these two different situations. Followed by this we will present Procedure 4.17 (which is a test generation procedure for case C(3.1)) which calls Subprocedure A (as a subroutine) when situation A is present and calls Subprocedure B when situation B is present.

Subprocedure A:

This procedure is called as a subroutine by Procedure 4.17 when situation A is present.

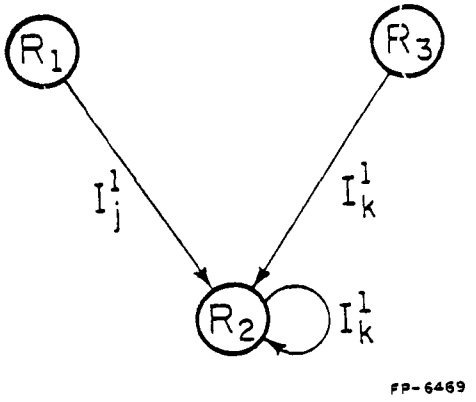


Figure 4.5. Illustrating Example 4.12.

Step 1: Execute I_j.

Step 2: Read out $D(I_j)$ by executing READ $(D(I_j))$.
/The p^{th} bit of the output data = x/

Step 3: If $\ell(S(I_k)) \geq 2$ then repeat steps 1 and 2 else go to step 4.

Step 4: Execute I_k.

Step 5: Read out $D(I_j)$ by executing READ $(D(I_j))$.
/The p^{th} bit of the output data = $\bar{x}$/ □

Subprocedure B:

This procedure is called as a subroutine by Procedure 4.17 when situation B is present.

Step 1: Execute I_k.

Step 2: Read out $D(I_k)$ by executing READ $(D(I_k))$.
/The p^{th} bit of the output data = $\bar{x}$/

Step 3: If $\ell(S(I_k)) \geq 2$ then repeat steps 1 and 2 else go to step 4.

Step 4: Execute I_j.

Step 5: Read out $D(I_j)$ by executing READ $(D(I_j))$.
/The p^{th} bit of the output data = x/ □

Procedure 4.17:

This procedure is applicable for case C(3.1) and case B(2.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = 2$ and $D(I_j) = D(I_k)$, and fault $f(I_j/I_k)$ when $\ell(I_j) = \ell(I_k) = 2$ and $D(I_j) = D(I_k)$.

```
FOR  i ← 1 TO 2 DO
BEGIN
```

Step 1: IF i = 1 THEN x ← 0 ELSE x ← 1.

Step 2: If possible, store proper operands in $S(I_j)$ and $S(I_k)$ such that if I_j is executed it produces x in some bit (say the p^{th} bit) of $D(I_j)$, and if I_k is executed it produces $\bar{x}$ in the p^{th} bit of $D(I_j)$.

Step 3: If step 2 requires x to be stored in the p^{th} bit of $D(I_j)$ then execute Subprocedure A else execute Subprocedure B.

END.

Note that if $D(I_j) \notin S(I_j)$ and $D(I_j) \notin S(I_k)$, then as mentioned earlier no specific data need to be stored in $D(I_j)$ in step 2 of Procedure 4.17. Therefore strictly speaking either of the subprocedures could be called in step 3 when $D(I_j) \neq S(I_j)$ and $D(I_j) \neq S(I_k)$.

Theorem 4.16: Procedure 4.17 detects $f(I_j/I_j+I_k)$ in case C(3.1) as well as $f(I_j/I_k)$ in case B(2.2).

Proof: In step 2, proper operands are chosen in $S(I_j)$ and $S(I_k)$ to produce x in some bit (say the p^{th} bit) of $D(I_j)$, if I_j is executed, and to produce $\bar{x}$ in the same bit of $D(I_j)$, if I_k is executed. If $f(I_j/I_j+I_k)$ is present, $x * \bar{x}$ will be produced in the p^{th} bit of $D(I_j)$, when I_j is executed. If $f(I_j/I_k)$ is present $\bar{x}$ will be produced in the p^{th} bit of $D(I_j)$. The first iteration of the FOR loop is for detecting the fault if * denotes the OR function, and the second iteration of the loop is for detecting the fault if * denotes the AND function.

Without loss of generality we assume that * denotes the OR function and situation A exists. Therefore in step 3, Subprocedure A is executed as a subroutine. If proper operands are really stored in $S(I_k)$ in step 2 of Procedure 4.17, the fault will be detected in step 2 of

Subprocedure A when $D(I_j)$ is read out. On the other hand, proper operands may not be stored in $S(I_k)$ due to faults involved in instructions used to write data in $S(I_k)$. In this case the fault will not be detected in step 2 of Subprocedure A. The microprocessor under test has not yet executed tests to detect $f(I_p/I_p+I_q)$ where $\ell(I_p) = 1$ and $\ell(I_q) \geq 3$. Therefore the execution of READ $(D(I_j))$ in step 2 of the subprocedure may cause additional execution of an instruction I_q such that $\ell(I_q) \geq 3$ and $D(I_q) \subseteq S(I_k)$. If this changes the contents of $S(I_k)$ the fault may be detected in step 3 of the subprocedure, otherwise it will be detected in step 5. (Refer to the details of the proof of Theorem 4.13.) □

Case C(3.2): $D(I_j) \neq D(I_k)$. In this case we follow the procedure below.

Procedure 4.18:

This procedure is applicable for case C(3.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = 2$ and $D(I_j) \neq D(I_k)$.

Step 1: Store proper operands in $S(I_k)$ and OPERAND 1 in $D(I_k)$ such that when I_k is executed it produces RESULT 1 in $D(I_k)$, and RESULT 1 ≠ OPERAND 1. If $\ell(S(I_k)) = 1$, read out $S(I_k)$ by executing READ $(S(I_k))$.

/To make sure that $S(I_k)$ contains expected operands/

Step 2: Execute I_j.

Step 3: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = OPERAND 1 ≠ RESULT 1/

Step 4: If $\ell(S(I_k)) \geq 2$ then repeat steps 2 and 3 else go to step 5.

Step 5: Execute I_k.

Step 6: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = RESULT 1/ □

Theorem 4.17: Procedure 4.18 detects $f(I_j/I_j+I_k)$ in case C(3.2).

Proof: We will give only the sketch of the proof since the basic ideas are essentially the same as used in the proofs of Theorem 4.10 through 4.17. If proper operands are really stored in $S(I_k)$ in step 1 the fault will be detected in step 3, since RESULT 1 will be read out. On the other hand, if proper operands are not stored in $S(I_k)$ due to faults in instructions used to write data in $S(I_k)$, the fault will be detected either in step 1 (if $\ell(S(I_k)) = 1$) when $S(I_k)$ is read out, or in step 4 or 6. □

4.3.4.4. Test Generation for $f(I_j/I_j+I_k)$ When $\ell(I_j) = \ell(I_k) = K \geq 3$

This case is referred to as case C(4). Note that according to the assumption in Section 4.3.1, instructions I_j and I_k belong to class I. When the microprocessor executes the tests to detect faults $f(I_j/I_j+I_k)$, where $\ell(I_j) = \ell(I_k) = K \geq 3$, it has already executed the tests to detect faults $f(I_p/I_p+I_q)$ where $1 \leq \ell(I_p) \leq K-1$ and $\ell(I_q) \leq 2$; in particular it has been already checked that the execution of any instruction of label $\leq K-1$ will not give rise to the additional execution of any instruction of class M. (Recall our assumption in Section 4.3.1 that if I_j belongs to class M, $\ell(I_j) \leq 2$.) We divide this case into two subcases depending on whether or not $D(I_j) = D(I_k)$.

Case C(4.1): $D(I_j) = D(I_k)$. Since $\ell(I_j) = \ell(I_k) = K$, $\ell(D(I_j)) = K-1$. This case is illustrated in Figure 4.4, where depending on $\ell(S(I_k))$ this case can be further divided into two subcases. Case C(4.1.1) refers to $\ell(S(I_k)) < K$, and case C(4.1.2) refers to $\ell(S(I_k)) = K$. Note that $\ell(S(I_k)) \not> K$, since $\ell(I_k) = K$ and $I_k \in$ class T. (Refer to the labeling algorithm given in Section 4.1.)

Case C(4.1.1): $\ell(S(I_k)) < K$. In this case we follow the procedure below.

Procedure 4.19:

This procedure is applicable for case C(4.1.1). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = K \geq 3$, $D(I_j) = D(I_k)$, and $\ell(S(I_k)) < K$. This procedure is essentially the same as Procedure 4.9 executed twice, with the following modifications: During the first execution of the procedure, in step 1 the condition to be satisfied by OPERAND 1 and OPERAND 2 is given by (OPERAND 1) $\vee$ (OPERAND 2) $\neq$ OPERAND 1, and during the second execution of the procedure the condition to be satisfied is given by (OPERAND 1) $\wedge$ (OPERAND 2) $\neq$ OPERAND 1.

Theorem 4.18: Procedure 4.19 detects $f(I_j/I_j+I_k)$ in case C (4.1.1).

Proof: The proof of this theorem parallels closely that of Theorem 4.8. We will, therefore, stress only those points where they differ. In step 2 of Procedure 4.19, I_j is executed and it is expected to produce OPERAND 1 in $D(I_j)$. If $f(I_j/I_j+I_k)$ exists, (OPERAND 1) * (OPERAND 2) would be produced instead, where as before * denotes the logical AND or OR function, provided $S(I_k)$ really contained OPERAND 2 when I_j is executed. In this case the fault will be detected in step 4. On the other hand,

due to the faults involved in the instructions used to write data in $S(I_k)$, OPERAND 2 may not be stored in $S(I_k)$; in this case the fault will be detected in step 3 itself. □

Case C(4.1.2): $\ell(S(I_k)) = K$. In this case we follow Procedure 4.10 twice, with the same modifications given for case C(4.1.1). We refer to this modified Procedure 4.10 as Procedure 4.20.

Theorem 4.19: Procedure 4.20 detects $f(I_j/I_j+I_k)$ in case C(4.1.2). □

The proof of this theorem follows closely those of Theorem 4.9 and 4.18, and hence is not repeated here.

Case C(4.2): $D(I_j) \neq D(I_k)$. Depending on $\ell(S(I_k))$ this case can be further divided into two subcases. Case C(4.2.1) refers to $\ell(S(I_k)) < K$, and case C(4.2.2) refers to $\ell(S(I_k)) = K$. Note that $\ell(S(I_k)) \not> K$, since $\ell(I_k) = K$ and I_k belongs to class T.

Case C(4.2.1): $\ell(S(I_k)) < K$. In this case we use the procedure below.

Procedure 4.21:

This procedure is applicable for case C(4.2.1). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = K \geq 3$, $D(I_j) \neq D(I_k)$, and $\ell(S(I_k)) < K$.

Step 1: Store OPERAND 1 in $S(I_k)$ and OPERAND 2 in $D(I_k)$, such that OPERAND 1 ≠ OPERAND 2.

Step 2: Read out $S(I_k)$ by executing READ $(S(I_k))$.

/Expected output data = OPERAND 1/

Step 3: Execute I_j.

Step 4: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = OPERAND 2 / □

Theorem 4.20: Procedure 4.21 detects $f(I_j/I_j+I_k)$ in case C(4.2.1).

Proof: Since $\ell(S(I_k)) < K$, no instruction in the READ $(S(I_k))$ sequence can have label greater than K-1. Also $\ell(D(I_k)) = K-1$, hence READ $(S(I_k))$ reads out $S(I_k)$ without routing its contents through $D(I_k)$. When the microprocessor under test executes this procedure, it has already executed the tests to detect $f(I_p/\phi)$, $f(I_p/I_q)$, and $f(I_p/I_p+I_q)$ where $1 \leq \ell(I_p), \ell(I_q) \leq K-1$, and $f(I_v/I_v+I_w)$ where $1 \leq \ell(I_v) \leq K-1$ and $\ell(I_w) = K$. Therefore, in step 2 of this procedure, $S(I_k)$ is correctly read out to make sure that it stores OPERAND 1, and it continues to store OPERAND 1 after READ $(S(I_k))$ is executed. In step 3, I_j is executed. If $f(I_j/I_j+I_k)$ is present, OPERAND 1 will be stored in $D(I_k)$ and the fault will be detected in step 4. □

Case C(4.2.2): $\ell(S(I_k)) = K$. In this case we follow the procedure below.

Procedure 4.22:

This procedure is applicable for case C(4.2.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $\ell(I_j) = \ell(I_k) = K \geq 3$, $D(I_j) \neq D(I_k)$, and $\ell(S(I_k)) = K$.

Step 1: Store OPERAND 1 in $S(I_k)$ and OPERAND 2 in $D(I_k)$, such that OPERAND 1 $\neq$ OPERAND 2.

Step 2: FOR i ← 1 TO K DO

BEGIN

Execute I_j;

Read out $D(I_k)$ by executing READ ($S(I_k)$).

/Expected output data = OPERAND 2/

END

Step 3: Execute I_k.

Step 4: Read out $D(I_k)$ by executing READ ($D(I_k)$).

/Expected output data = OPERAND 1 /

Theorem 4.21: Procedure 4.22 detects $f(I_j/I_j+I_k)$ in case C(4.2.2). □

The proof of this theorem follows very closely that of Theorem 4.9 and is not repeated here.

4.3.4.5. Test Generation for $f(I_j/I_j+I_k)$ When $1 \leq \ell(I_j) \leq K$, $\ell(I_k) = K+1$, and $K \geq 2$.

This case is referred to as case C(5). Note that I_k belongs to class T because $\ell(I_k) = K+1$ and $K \geq 2$. Since $\ell(I_j) \leq K$ and $\ell(I_k) = K+1$, $\ell(D(I_j)) \leq K-1$ and $\ell(D(I_k)) = K$; hence $D(I_j) \neq D(I_k)$. We divide this case into two subcases depending on whether the value of $\ell(S(I_k))$ is less than K or is equal to K or K+1. Note that it cannot be greater than K+1 because $\ell(I_k) = K+1$ and I_k belongs to class T.

Case C(5.1): $\ell(S(I_k)) \leq K-1$. We follow Procedure 4.21 in this case. Following the arguments similar to those given in the proof of Theorem 4.20, it can be proved that Procedure 4.21 also detects $f(I_j/I_j+I_k)$ in case C(5.1).

Case C(5.2): $\ell(S(I_k)) = K$ or $K+1$. We follow Procedure 4.22 in this case. Following the arguments similar to those in the proof of Theorem 4.9, it can be proved that Procedure 4.22 also detects $f(I_j/I_j+I_k)$ in case C(5.2).

4.3.4.6. Test Generation for $f(I_j/I_j+I_k)$ When $K+1 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = K$.

This case is referred to as case C(6) and is divided into three subcases depending on the value of K.

Case C(6.1): $2 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = 1$. In this case the fault $f(I_j/I_j+I_k)$ will be readily detected due to the highest observability of I_k. For example, during the execution of I_k, data is transmitted between the microprocessor and the main memory or an I/O device, or the logic values on certain status pins are changed (e.g., during the execution of the "Interrupt enable" instruction). Such is not the case during the execution of I_j, therefore $f(I_j/I_j+I_k)$ will be readily detected when I_j is executed.

Case C(6.2): $3 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = 2$. The following procedure is followed in this case.

Procedure 4.23:

This procedure is applicable for case C(6.2). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $3 \leq \ell(I_j) \leq K_{max}$, and $\ell(I_k) = 2$.

Step 1: Store proper operands in $S(I_k)$ and OPERAND 1 in $D(I_k)$ such that when I_k is executed RESULT 1 is produced in $D(I_k)$, and OPERAND 1 $\neq$ RESULT 1. If $D(I_j) \subseteq S(I_k)$ then store a proper operand in $S(I_j)$ such that when I_j is executed the contents of $D(I_j)$ will not change. /To ensure that after executing I_j,

the contents of $S(I_k)$ remain unchanged/

Step 2: Execute I_j.

Step 3: Read out $D(I_k)$ by executing READ ($D(I_k)$).

/Expected output data = OPERAND 1/

Step 4: If $\ell(S(I_k)) \geq 3$ then repeat steps 2 and 3 else go to step 5.

Step 5: Execute I_k.

Step 6: Read out $D(I_k)$ by executing READ ($D(I_k)$).

/Expected output data = RESULT 1/ □

Theorem 4.22: Procedure 4.23 detects $f(I_j/I_j+I_k)$ in case C(6.2).

The proof of this theorem is not given as it follows the same arguments given in the proofs of Theorems 4.10 through 4.17.

Case C(6.3): $K+1 \leq \ell(I_j) \leq K_{max}$, $\ell(I_k) = K \geq 3$. Therefore according to the assumption in Section 4.3.1, instructions I_j and I_k belong to class T. In this case the procedure given below is followed.

Procedure 4.24:

This procedure is applicable for case C(6.3). It generates tests to detect fault $f(I_j/I_j+I_k)$ when $K+1 \leq \ell(I_k) \leq K_{max}$, $\ell(I_k) = K$, and $K \geq 3$.

Step 1: Store OPERAND 1 in $S(I_k)$ and OPERAND 2 in $D(I_k)$ such that OPERAND 1 $\neq$ OPERAND 2. If $D(I_j) = S(I_k)$ then store OPERAND 1 in $S(I_j)$. /To ensure that after executing I_j, the contents of $S(I_k)$ remain unchanged/

Step 2: Execute I_j.

Step 3: Read out $D(I_k)$ by executing READ $(D(I_k))$.

/Expected output data = OPERAND 2. $S(I_k)$ would continue to store its data after executing READ $(D(I_k))$, since $\ell(S(I_k)) \leq K$/

Step 4: Execute I_k.

Step 5: Read out $D(I_k)$ by executing READ $(D(I_k))$

/Expected output data = OPERAND 1/ □

Theorem 4.23: Procedure 4.24 detects $f(I_j/I_j+I_k)$ in case C(6.3). □

The proof of this theorem is not given as it follows the same arguments used in the proofs of Theorems 4.10 through 4.17.

4.4. Test Generation Procedures for Detecting Faults in the Data Transfer Function and the Data Storage Function

We motivate the discussion by means of an example. Let $I_{j_1}, I_{j_2}, \ldots, I_{j_k}$ be a sequence of instructions of class T such that $E(I_{j_1}), E(I_{j_2}), \ldots, E(I_{j_k})$ form a directed path from the IN node to the OUT node in the corresponding S-graph. Let the transfer paths in sets $T(I_{j_1}), T(I_{j_2}), \ldots, T(I_{j_k})$ each be w lines in width. Figure 4.6 illustrates the notation. We propose Procedure 4.25 to detect any fault in $T(I_{j_1}), T(I_{j_2}), \ldots, T(I_{j_k})$, and in registers $D(I_{j_1}), D(I_{j_2}), \ldots, D(I_{j_{k-1}})$. The fault models for the data storage function and data transfer function are given in Section 3.3 and 3.4, respectively.

Procedure 4.25:

I_{j_1} with data 111...1 ; /Write $D(I_{j_1})$ with data 111...1/

(w width) (w width)

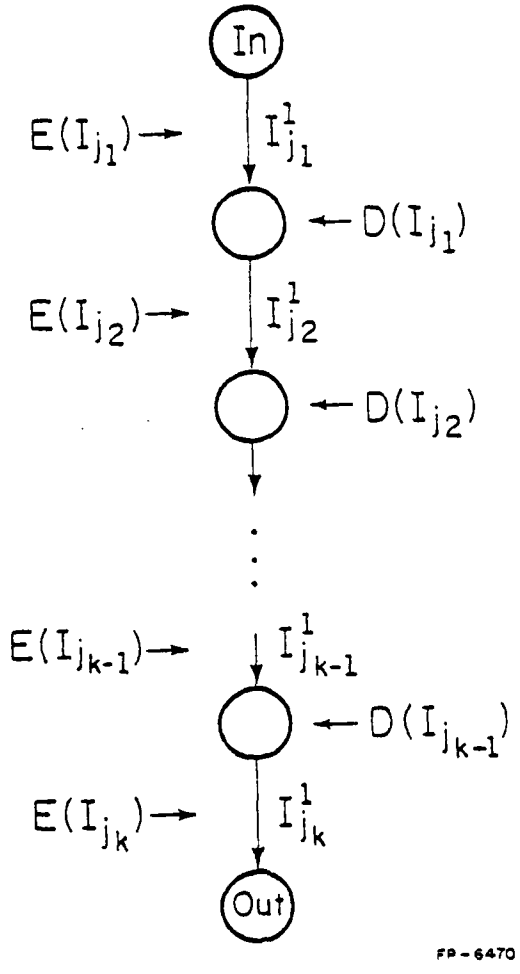


Figure 4.6. Illustrating the notation used in Procedure 4.25.

I_{j_2}; I_{j_3}; ...; I_{j_k}; /Expected output data = 111...1/ (w width)

I_{j_1} with data 11...1 00...0 /Write $D(I_{j_1})$ with data 11...1 00...0/ (w/2 w/2 w/2 w/2)

I_{j_2}; I_{j_3}; ...; I_{j_k}; /Expected output data = 11...1 00...0/ (w/2 w/2)

I_{j_1} with data 11...1 00...0 11...1 00...0 ; (w/4 w/4 w/4 w/4)

/Write $D(I_{j_1})$ with data 11...1 00...0 11...1 00...0/ (w/4 w/4 w/4 w/4)

I_{j_2}; I_{j_3}; ...; I_{j_k}; /Expected output data = 11...1 00...0 11...1 00...0/ (w/4 w/4 w/4 w/4)

.
.
.
.

I_{j_1} with data 1010...10; /Write $D(I_{j_1})$ with data 1010...10/ (w)

I_{j_2}; I_{j_3}; ...; I_{j_k}; /Expected output data = 1010...10/ (w)

Repeat the instructions above with complementary data. □

Theorem 4.24: Procedure 4.25 detects

1) a line in any transfer path in the set $T(I_{j_1}) \cup T(I_{j_2}) \cup \ldots \cup T(I_{j_k})$ stuck at 0 or 1.

2) two or more lines in any transfer path in the set $T(I_{j_1}) \cup T(I_{j_2}) \cup \ldots \cup T(I_{j_k})$ coupled.

3) a cell of any register in the set $D(I_{j_1}) \cup D(I_{j_2}) \cup \ldots \cup D(I_{j_{k-1}})$ stuck at 0 or 1.

Proof: If a fault is described by 1 or 3 above, it will be detected either after the execution of sequence $I_{j_2}, I_{j_3}, \ldots, I_{j_k}$ when the expected output data = 111...1, or after the execution of the sequence when the expected output data = 000...0.

Procedure 4.25 also detects any fault described by 2, because at some stage of the procedure any given two lines of a transfer path are required to carry different logic values i.e., x and $\bar{x}$, $x \in \{0,1\}$, respectively. If these two lines are coupled they will fail to carry different logic values, and the fault will be detected after the subsequent execution of the sequence $I_{j_2}, I_{j_3}, \ldots, I_{j_k}$. □

We define the set of tests in Procedure 4.25 "transfer test set" and the associated data being routed on the corresponding transfer paths "transfer test data". Consider the transfer paths associated with the instructions of class T. Concentrate on the subgraph of the S-graph that represents instructions of class T only. We call this subgraph the T-subgraph. Let P_1 be a directed path from the IN node to the OUT node in the T-subgraph. All the instructions which are represented by edges constituting path P_1 are said to be covered by path P_1. Let $\{P_1, P_2, \ldots, P_n\}$ be a set of directed paths from the IN node to the OUT node of the T-subgraph such that this set collectively covers all the instructions of class T. It is clear from Theorem 4.24 that if the transfer test data is routed from the IN node to the OUT node using the transfer test sets consisting of instructions covered by each path in set $\{P_1, P_2, \ldots P_n\}$ any fault in the transfer paths associated with the instructions of class T, or any fault in the data storage function will be detected (since the transfer test data is "routed" on every edge of the T-subgraph, every node of the graph

is also visited).

We need to test the transfer paths associated with the instructions of class M. Let "R_i o $R_j \rightarrow R_k$" denote a typical instruction of class M which performs an operation denoted by "o" on the contents of register R_i and R_j, and stores the result in register R_k. We need to choose a set of proper operands in R_i and R_j such that when the instruction "R_i o $R_j \rightarrow R_k$" is executed it generates and stores data corresponding to the transfer test set in R_k. We then need to route these data from R_k to the OUT node by executing READ (R_k). This test ensures that the transfer path from the data manipulation logic (e.g. the ALU used in executing instruction "R_i o $R_j \rightarrow R_k$") to register R_k is fault free. Therefore any result generated by this instruction can be faithfully transferred to R_k. Consider instruction I_4 in the S-graph of Figure 2.8. We execute Procedure 4.26 to test the transfer path from the ALU to R_1. (It is assumed that the microprocessor is an 8-bit processor.)

Procedure 4.26:

I_1 with data 1111 1111 ;

I_2 with data 0000 0000 ;

I_4 ;

I_7 ;

Repeat the tests above with data (1111 0000, 0000 0000), (1100 1100, 0000 0000), (1010 1010, 0000 0000), (0000 0000 0000 0000), (0000 1111, 0000 0000), (0011 0011, 0000 0000), (0101 0101, 0000 0000). □

We also need to check that the transfer paths connecting R_i and R_j to the ALU in the "R_i o $R_j \rightarrow R_k$" instruction are fault free. This

ensures that any pair of operands can be applied to the ALU in the "R_i o $R_j \rightarrow R_k$" instruction. For this we need to check that any line in the transfer paths from R_i and R_j to the ALU can be set to 0 or 1 independent of the logic values on any other line in these transfer paths. Consider instruction I_4 in Figure 2.8. We want to test the transfer paths connecting R_1 and R_2 to the ALU. We execute Procedure 4.27.

Procedure 4.27:

```
I1  with data 0000 0001 ;   /R1 stores 0000 0001/
I2  with data 0000 0000 ;   /R2 stores 0000 0000/
I4 ;
I7 ;                        /Expected output data 0000 0001/
I1  with data 0000 0010 ;   /R1 stores 0000 0010/
I2  with data 0000 0000 ;   /R2 stores 0000 0000/
I4 ;
I7 ;                        /Expected output data 0000 0010/
.
.
.
.
.
I1  with data 1000 0000 ;   /R1 stores 1000 0000/
I2  with data 0000 0000 ;   /R2 stores 0000 0000/
I4 ;
I7 ;                        /Expected output data 1000 0000/
I1  with data 0000 0000 ;   /R1 stores 0000 0000/
I2  with data 0000 0001 ;   /R2 stores 0000 0001/
I4 ;
I7 ;                        /Expected output data 0000 0001/
```

I_1 with data 0000 0000 ; /R_1 stores 0000 0000/

I_2 with data 0000 0010 ; /R_2 stores 0000 0010/

I_4 ;

I_7 ; /Expected output data 0000 0010/

.
.
.
.

I_1 with data 0000 0000 ; /R_1 stores 0000 0000/

I_2 with data 1000 0000 ; /R_2 stores 1000 0000/

I_4 ;

I_7 ; /Expected output data 1000 0000/

Repeat the test above with complementary data. □

We need to execute tests similar to those given in Procedures 4.26 and 4.27 for every instruction of class M. Finally we must test the transfer paths associated with the instructions of class B. This is accomplished by choosing the set of jump or branch addresses such that they correspond to the transfer test set for jump, branch, return from subroutine, etc., instructions. For example, in order to test the transfer path associated with the jump instruction (instruction I_9) in Figure 2.8 we need to execute Procedure 4.28. It is assumed that the width of the address bus is 16.

Procedure 4.28:

I_9 with jump address 0000 0000 0000 0000 ;

I_9 with jump address 0000 0000 1111 1111 ;

I_9 with jump address 0000 1111 0000 1111 ;

I_9 with jump address 0011 0011 0011 0011 ;

I_9 with jump address 0101 0101 0101 0101 ;

Repeat the test above with the complementary set of jump addresses. □

4.5. Test Generation Procedure for Detecting Faults in the Data Manipulation Function

As described in Section 3.5 we assume that complete test sets are available for detecting faults (for some specified fault model) in the ALU and other functional units such as a shifter, logic used to increment the program counter, or the interrupt handling logic. The operands specified by such test sets can be provided to a functional unit using, in general, a sequence of instructions of class T. Similarly, the result produced by a functional unit can be read out using a sequence of instructions of class T.

If the logic level description of functional units is available, test sets can be generated for them using classical fault detection algorithms based on the stuck-at fault model. On the other hand, if we do not know the logic level details of the ALU, but know, for example, that it is realized using an iterative logic array we can generate test sets for it as explained in [Dias76]. Another approach would be to generate test sets for functional testing of the ALU, shift, increment, compare logic, etc., using binary decision diagrams [Aker78].

Even though some faults associated with the instruction decoding and control function look like faults in the data manipulation function, and vice versa, the set of faults in one function is not a subset or superset of the set of faults in the other function. For example, under a fault in the instruction decoding function an "Add" instruction may be decoded as a "Subtract" instruction. This fault cannot be distinguished from a fault in the ALU: however, if the test procedure

for detecting faults in the instruction decoding and control function (presented in Section 4.3) is executed correctly, it cannot guarantee the absence of faults in the ALU. In order to detect a fault in the ALU or any other functional units we need to execute the corresponding test sets. Similarly if the test procedure for detecting faults in the data manipulation function is executed correctly, it does not guarantee the absence of faults in the instruction decoding and control function. For example, whenever an "Add" instruction is executed, it may additionally activate an instruction that complements a certain register not involved in the "Add" instruction. Such a fault can be detected by the test procedure used to detect faults in the instruction decoding and control function and not by the test procedure used to detect faults in the data manipulation function.

5. COMPLEXITY OF THE TEST SEQUENCES

We now determine the complexity of the test sequences generated by various procedures given in Chapter 4. The complexity is measured in terms of the number of instructions generated as a function of n_R - the number of registers in set $\mathcal{R}$, or n_I - the number of instructions in the instruction repertoire. This will help in exploring the relationship between the architecture of a microprocessor and the complexity of the test sequences.

Theorem 5.1: The worst case complexity of the test sequence generated by Procedure 4.1 is $\mathcal{O}(n_R^3)$, where $|\mathcal{R}| = n_R$.

Proof: Let us consider the number of instructions that are generated in steps 3(a), (b), and (c) of Procedure 4.1 when there are K registers in set A, i.e., $|A| = K$. As a result of the labeling algorithm, $\max(\ell(R_i)) = K' \leq K$ for every register R_i of set A. Also, every number in the integer set $\{1,2, \ldots, K'\}$ is assigned as a label to at least one register in set A. Therefore, in the worst case, during step 3(b) we need to generate $\Sigma(1 + 2 + \ldots + K)$ instructions which would read out the registers of set A, where $|A| = K$. Since set A is augmented only by one register during each iteration of step 3, in the worst case, $\sum_{k=1}^{n_R-1} \Sigma(1 + 2 + \ldots + K)$ instructions need to be performed for reading out registers of set A. Similarly, in all iterations of step 3(c), a total of $\sum_{K=1}^{n_R-1} (K + 1)$ instructions need to be generated to read out the register at the front of the queue.

Since there are only n_R registers, the register that is "farthest away" from the IN node can be written by executing at most n_R instructions of class T. Moreover, if this register needs n_R instructions

to be written into, the "next farthest away" register will require (n_R-1) instructions, and so on. Therefore when $|A| = K$, in the worst case, $n_R-(K-1) + n_R-(K-2) + (n_R-1) + n_R$ instructions are required to write the registers of a set A during step 3(a). During all iterations of step 3(a), a total of $\sum_{K=1}^{n_R-1} \sum n_R-(K-1) + n_R-(K-2) + \ldots + n_R$ instructions are needed to write the registers of set A. Similarly in all iterations of step 3(a), a total of $\sum_{K=1}^{n_R-1} (n_R-K)$ instructions are needed to write the register at the front of the queue, in the worst case. When all the terms involved in series are summed up, a total of $n_R^3 + 2n_R^2 - n_R - 2$ instructions are generated. Hence the worst case complexity of the test sequence is $\mathcal{O}(n_R^3)$. □

It is instructive to illustrate the worst case example which is shown by a partial S-graph in Figure 5.1. Note that $\mathcal{O}(n_R^3)$ complexity is only for the worst case. For $n_R = 7$, 432 instructions are generated in the worst case. However, for the example microprocessor (where $n_R = 7$) only 2 x 53 = 106 instructions are generated as shown in Example 4.2. The exact number of instructions depends on the distribution of integer labels of the nodes. In fact, if all nodes have label 1 (i.e., if the architecture contains only the so-called accumulators and general purpose registers which can be directly loaded into and stored from the main memory), the complexity of the test sequence would be $\mathcal{O}(n_R^2)$. On the other hand, if the architecture allows scratch-pad registers and on-chip LIFO stacks, for example, (giving rise to nodes with labels greater than 1 in the S-graph), the complexity of the test sequence approaches $\mathcal{O}(n_R^3)$.

It is very difficult to find a closed form solution for the worst case complexity of the test sequence generated by procedures in

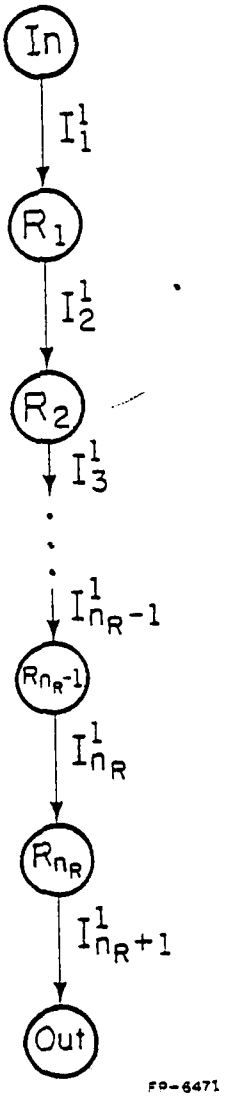


Figure 5.1. The worst-case example requiring $\Theta(n_R^3)$ instructions to be generated by Procedure 4.1.

Section 4.3. We will therefore concentrate on the dominating term in the worst case complexity calculation. This dominating term can be attributed to the loop in step 2 of Procedures 4.10, 4.20 and 4.22. Let n_{I_i} be the number of instructions whose edge sets have been labeled i in the S-graph. As denoted earlier, let K_{max} be the maximum value of labels associated with the edge sets. Thus $\sum_{i=1}^{K_{max}} n_{I_i} = n_I$. The dominating term accounts for $\sum_{i=1}^{K_{max}} n_{I_i}(i^i)$ instructions generated by step 2 of Procedures 4.10, 4.20 and 4.22.

If the architecture allows instructions which are represented by edge sets with labels much greater than 1, the length of the test sequence could become very large in the worst case, since the complexity grows exponentially (note the i^i factor in the expression above). This is because instructions represented by edge sets with large labels impart very poor observability to the architecture, i.e., a large number of instructions need to be executed to read out internal registers; this is reflected in the increased length of the test sequences generated by procedures in Section 4.3. However, the expression above is applicable only in the worst case; in many typical architectures $K_{max} \leq 3$, deemphasizing the effect of the dominating term. In fact, if $K_{max} \leq 2$ (i.e., the instruction repertoire contains instructions that store their result in the main memory or the accumulators and general-purpose registers), Procedures 4.10, 4.20 and 4.22 will not be required at all. In the case of such architectures, the complexity of the test sequences generated by the procedures given in Section 4.3 can be approximated to $\mathcal{O}(n_I^2)$ because there are $\mathcal{O}(n_I^2)$ faults (in the instruction decoding and control function) and none of them will require Procedure 4.10, 4.20 or 4.22 to generate tests for it. Note that no other procedure has a loop similar to that in step 2 of Procedures 4.10, 4.20

and 4.22. Therefore the complexity of test sequences generated by these procedures can be approximated to $\mathcal{O}(n_I^2)$.

The length of the test sequence generated by various procedures given in Section 4.4 depends on the widths of data and address buses, the nature of operation "o" performed by instructions "R_i o $R_j \rightarrow R_k$" of class M, and the distribution of integer labels associated with edge sets, i.e., n_{I_i} and K_{max}. If there are many instructions with higher labels (i.e., large n_{I_i} for larger i), the length of the test sequences required to detect a fault in the data transfer and data storage functions increases.

For today's microprocessors n_R typically ranges from 4 to 32, while n_I ranges from 30 to 200. Note that the complexity of the test sequences for detecting faults in the instruction decoding and control function is at least $\mathcal{O}(n_I^2)$, while the complexity of the test sequences for detecting faults in the register decoding function is between $\mathcal{O}(n_R^2)$ and $\mathcal{O}(n_R^3)$. Therefore the test sequences used to detect faults in the instruction decoding and control function constitute a dominant portion of the test sequences for a microprocessor.

6. A CASE STUDY

Test sequences were generated for a real microprocessor by applying the test generation procedures developed in the thesis. The goal of the study was two-fold. First, we wanted to generate the test sequences to gain insight into problems involved in using the test generation procedures. We believe that this is an essential first step towards automating the test generation procedures which will operate on a given S-graph. Secondly, we wanted to evaluate the fault coverage of the test sequences for stuck-at faults for a real microprocessor.

A microprocessor from the Hewlett-Packard Company was used. The HP microprocessor is a single chip, n-channel MOS, 8 bit parallel, control oriented processor. All instructions and data are transferred in and out of the microprocessor with an 8 bit bidirectional data bus. Program addresses are transferred out on an 11 bit address bus. There can be up to 15 I/O ports. The normal program may be interrupted by use of the interrupt request control line. The interrupt scheme is fully vectored with 256 possible vectors. The processor can control external circuits and check their status through the use of 7 bidirectional control lines.

The microprocessor contains one 8 bit accumulator, one control logic unit, one 1 bit extend register, sixteen 8 bit registers, one 8 bit magnitude comparator, one 11 bit program counter, one 11 bit subroutine stack register, and one 11 bit interrupt stack register. The instruction set has 187 instructions that includes instructions transferring data between the memory and the accumulator, between the memory and (sixteen 8 bit) registers, between the accumulator and registers, between the accumulator and I/O devices, instructions performing bit manipulations and magnitude comparisons, instructions performing conditional and unconditional jumps in the program sequencing, etc.

We adopted the following strategy in applying the test sequences. Since, as shown in Chapter 5, the test sequences for the instruction decoding and control function form the dominant portion of the test sequences for the microprocessor, we first applied the test sequences for the register decoding function, the data transfer and the data storage function, and the data manipulation function. The length of these sequences was approximately 1 K instructions. This was followed by application of the test sequences for the instruction decoding and control function. The length of these sequences was approximately 8 K instructions. (Recall that there are 187 instructions in the instruction repertoire.) The test sequences were generated by using only the information about the instruction set and organization of the microprocessor.

In order to determine the fault coverage of the test sequences for stuck-at faults, a detailed gate and subnetwork model of the microprocessor (obtained from Hewlett-Packard) was used on the TESTAID III fault simulator. Approximately 2200 single stuck-at faults were simulated. The test sequences generated were run in segments (since the simulator could not handle all the tests at one time) and the fault coverage of each segment was noted. The test sequences for the register decoding, data storage, data transfer, and data manipulation functions were able to detect about 90% of all single stuck-at faults. About 6% of the faults gave rise to simultaneous execution of multiple instructions as described by the fault model for the instruction decoding and control function. Many of these faults were subtle and difficult to detect and very elaborate test sequences were required (accounting for 8 K instructions). For example, when executing the instruction "Skip if the n^{th} bit of the accumulator is 0" (with n between 0 and 7), under a particular single-at fault, the above

instruction will be executed correctly, but at the same time, the contents of the accumulator are also stored in the n^{th} register. Some examples of $f(I_j/I_j+I_k)$ faults found in the case study are given in Table 6.1. (the table lists instructions I_j and I_k.)

The remaining faults (about 4%) were associated with the power-up and initialization logic, or were undetectable because of redundancies in the logic, or required invalid opcodes to detect them. Thus for this particular microprocessor the fault coverage was excellent.

The test generation effort was quite straightforward and we believe that it can be automated without much difficulty. The overall results of the case study were quite promising and we are convinced that our approach is a viable and effective one for generating tests for microprocessors.

Table 6.1. Instructions I_j and I_k for which fault $f(I_j/I_j+I_k)$ exists.

Instruction I_j	Instruction I_k
Clear the extend bit.	Transfer the contents of the accumulator to register R_5.
Return from interrupt and enable interrupt.	Transfer the contents of the accumulator to register R_1.
Clear the third control flag.	Disable interrupt.
Skip if the first control flag is zero.	Enable interrupt.
Clear the first bit of the accumulator.	Clear the zeroth bit of the accumulator.

7. CONCLUDING REMARKS

7.1. Summary of Thesis

The purpose of this research has been to develop test generation procedures for testing microprocessors that would treat the microprocessor organization and instruction set as parameters. The test generation effort is assumed to be in a user environment where the gate and flip-flop level details of the microprocessor are not known. The procedures will generate tests which can be assembled into valid machine instructions. The microprocessor under test executes these instructions which are stored in the memory of an external tester which continually monitors all the input and output pins of the microprocessor. A fault is detected when the data on any output pin is different from the expected data.

In Chapter 2, the instruction repertoire of the microprocessor was divided into three classes (T, M, and B). Then a graph-theoretic model for microprocessor (called the S-graph) was developed. Each register is represented by a node in the S-graph and data flow involved during the execution of an instruction is represented by a set of directed edges. The motivation behind this approach was to be able to construct a model for the microprocessor for test generation purposes using only the information available in the typical user's manual. This is because the gate and flip-flop level information needed to construct a model at the logic level is not only unavailable, but classical test generation methods which go hand in hand with the logic level model will be very complicated and expensive due to the very large number of gates and flip-flops on the microprocessor chip.

Functional level fault models describing faulty behavior in the register decoding function, instruction decoding and control function, data

transfer function, and data manipulation function were presented in Chapter 3. Various underlying fault mechanisms responsible for functional level faults were pointed out. The fault models are quite independent of the details of implementation. The effects of faults on the behavior of the microprocessor were described at the level of the S-graph.

In Chapter 4, test generation procedures were given to detect faults in the fault models. The first step in test generation is to assign integer labels to the nodes and edges of the S-graph by using the labeling algorithm given in Section 4.1. The label assigned to a node indicates the shortest "distance" of that node to the OUT node (in terms of the instructions of class T or B); the label assigned to the edge set representing an instruction is directly derived from the label assigned to its destination register.

Test generation procedures presented in the subsequent sections of the chapter take full advantage of the information obtained from these labels; tests are generated in such a way that the knowledge gained from the correct execution of tests used for checking the decoding of registers and instructions with lower labels is utilized in generating tests for checking the decoding of registers and instructions with higher labels.

In Chapter 5, the complexity of test sequences generated by the test generation procedures in Chapter 4 was studied. The complexity is measured in terms of the number of instructions generated as a function of n_R- the number of instructions in the instruction repertoire. The worst case complexity of the test sequence for the register decoding function was shown to be $\mathcal{O}(n_R^3)$; however, if all registers have label 1 (indicating the highest observability) the complexity would be $\mathcal{O}(n_R^2)$.

It was shown that if the instructions have labels less than or equal to two, the complexity of test sequences for the instruction decoding and control function is $\mathcal{O}(n_I^2)$. If the architecture allows instructions with labels greater than two, the complexity increases very rapidly. Since n_R typically ranges from 4 to 32, while n_I ranges from 30 to 200, the test sequences for the instruction decoding and control function constitute a dominant part of the test sequences of a microprocessor.

In Chapter 6 we have described our effort regarding the development of test sequences based on the test generation procedures in Chapter 4 for a real 8-bit microprocessor from the Hewlett-Packard Company. Approximately 2200 single stuck-at faults were simulated. About 96% of these faults were detected by these test sequences. The remaining faults were associated with the power-up and initialization logic, or were undetectable because of redundancies in the logic or they required invalid opcodes for their detection. The results of our study were quite promising.

Thus to summarize the thesis, our approach allows us to treat the organization and instruction set of microprocessors as parameters of the test generation procedures. The information needed to construct the S-graph is easily available in the user's manual. We believe our approach is a viable and effective one towards generating test sequences for microprocessors.

7.2. Suggested Future Research

The S-graph of the microprocessor is capable of modeling most of the architectural features observed with current microprocessors. However, it cannot model some of the features observed in the new, powerful 16-bit microprocessors. For example, instructions exchanging data among two

register files cannot be adequately modeled. In order to understand the effects of these architectural features on test generation, further research needs to be done to model these architectural features using the S-graph or some other similar technique.

Fault model for the instruction decoding and control function considers only "gross" faults $f(I_j/\phi)$, $f(I_j/I_k)$, and $f(I_j/I_j+I_k)$. Our case study regarding the test generation for the Hewlett-Packard microprocessor and the subsequent evaluation of the fault coverage showed that this fault model was adequate to account for all single stuck-at faults in the instruction decoding and control function of this particular microprocessor. We do not know how adequate the fault model would be for other microprocessors, particularly the new 16-bit microprocessors. (Some of them have an on-chip microprogrammed control unit.) Further research needs to be directed towards evaluating the necessity of modeling other faults such as the ones that give rise to partial execution of an instruction, or a change in the sequence of data flow involved in an instruction. Furthermore, if the evaluation study points to the necessity of the improved fault models, the next problem will be to describe the effects of these faults at the level of the S-graph and then develop test generation procedures to detect these faults.

Another important area of future research is to study the applicability of the test sequences generated by the procedures of Chapter 4 in a testing environment where the sophisticated tester required by our approach cannot be used. For example, in field testing, the external tester must be very simple and most of the testing tasks (such as comparing the output result on a bus with the expected result) must themselves be

carried out by the microprocessor under test. Future research must be directed towards investigating the modifications to the proposed test generation procedures to make them suitable for field testing, or for the so-called "self-testing" operations. Self-testing involves some hardcore, i.e., that part of hardware which must be assumed to be fault free. Therefore identification of the hardcore and its testing by an external tester are two major problems that need to be solved for any self-testing scheme.

Finally, future research needs to be directed towards the challenging problem of design for testability. Architectural features which enhance testability should be investigated. Allowing registers and instructions with as low labels as possible (imparting high observability) is obviously a step in the right direction. Such a solution might degrade the performance of the microprocessor in yet unknown way. The underlying testability-performance trade-offs should also be investigated.

REFERENCES

[Aker78] Akers, S. B., "Functional testing with binary decision diagrams," in Proc. of the 8th International Conference on Fault-Tolerant Computing, Toulouse, France, IEEE Computer Society, pp. 75-82, June 1978.

[Ande76] Anderson, R. E., "Microprocessor test mehtods change to meet complex demands," Electronics, pp. 125-128, April 15, 1976.

[BaKi76] Batni, R. P. and C. R. Kime, "Module-level testing approach to combinational circuits," IEEE Trans. on Computers, Vol. C-25, pp. 594-604, June 1976.

[Ball79] Ballard, D. R., "Designing fail-safe microprocessor systems," Electronics, pp. 139-143, January 4, 1979.

[BrFr76] Breuer, M. A. and A. D. Friedman, Diagnosis and Reliable Design of Digital Systems, Computer Science Press, Inc., 1976.

[ChMc76] Chiang, A. C. L. and R. McCaskill, "Two new approaches simplify testing of microprocessors," Electronics, pp. 100-105, January 22, 1976.

[CMMe70] Chang, H. Y., E. G. Manning, and G. Metze, Fault Diagnosis of Digital Systems, Wiley-Interscience, New York, 1970.

[Cush77] Cushman, R. H., "Fourth annual microprocessor directory," EDN, Vol. 22, pp. 44-83, November 20, 1977.

[Dias76] Dias, F. J. O., "Truth-table verification of an iterative logic array," IEEE Trans. on Computers, Vol. C-25, pp. 605-613, June 1976.

[FeeW78] Fee, W. G., "Tutorial LSI testing," Second Edition, IEEE Computer Society, IEEE Catalog No. EHO 122-2, 1978.

[Flyn74] Flynn, M. J., "Trends and problems in computer organizations," IFIP Proceedings 1974, North-Holland Publishing Company, pp. 3-10.

[GsMc75] Gschwind, H. W. and E. J. McCluskey, Design of Digital Computers, Springer-Verlag, New York, Inc., pp. 355-366, 1975.

[Haye76] Hayes, J. P., "Transition count testing of combinational logic circuits," IEEE Trans. on Computers, Vol. C-25, pp. 613-620, June 1976.

[HPJO77] "Signature analysis," Hewlett-Packard Journal, Vol. 28, No. 9, May 1977.

[Hust74] Huston, R., "Microprocessor function test generation on the Sentry 600," Fairchild Systems Technical Bulletin 4, November 1974.

[INTE75] Intel 8080 Microcomputer Systems User's Manual, September 1975, Santa Clara, California.

[Koha70] Kohavi, Z., Switching and Finite Automata Theory, Chapter 13, McGraw-Hill Book Company, 1970.

[LiDo79] Lippman, M. D. and E. S. Donn, "Design forethought promotes easier testing of microcomputer boards," Electronics, pp. 113-119, January 18, 1979.

[Mann66] Manning, E., "On computer self-diagnosis: Part I and II," IEEE Trans. on Computers, Vol. EC-15, pp. 873-890, December 1966.

[NTAb78] Nair, R., S. M. Thatte, and J. A. Abraham, "Efficient algorithms for testing semiconductor random-access memories," IEEE Trans. on Computers, Vol. C-27, pp. 572-576, June 1978.

[Powe69] Powell, T. S, "A module diagnostic procedure for combinational logic," Coordinated Science Laboratory Report R-413, University of Illinois, Urbana, Illinois, April 1969.

[RBSc67] Roth, J. P., W. G. Bouricius, and P. R. Schneider, "Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits," IEEE Trans. on Computers, Vol. C-20, pp. 1286-1293, November 1971.

[RoSa75] Robach, C. and G. Saucier, "Diversified test methods for local control units," IEEE Trans. on Computers, Vol. C-24, pp. 562-567, May 1975.

[RoSa78] Robach, C. and G. Saucier, "Dynamic testing of control units," IEEE Trans. on Computers, Vol. C-27, pp. 617-623, July,1978.

[Scol72] Scola, P., "An annotated bibliography of test and diagnostics," Honeywell Computer Journal, Vol. 6, 1972, pp. 97-102, and 105-162.

[TEST75] Various papers on microprocessor testing in the Digests of Semiconductor Test Symposiums, Cherry Hill, N.J., IEEE Computer Society, Oct. 14-16, 1975, Oct. 17-21, 1976, Oct. 25-27, 1977.

[ThAb78] Thatte, S. M. and J. A. Abraham, "A methodology for functional level testing of microprocessors," in Proc. of the 8th International Conference on Fault-Tolerant Computing, Toulouse, France, IEEE Computer Society, pp. 90-95, June 1978.

APPENDIX

We consider the behavior of a decoder (for a given valid input) under a single stuck-at fault. The decoder is assumed to be realized without any reconvergent fanout. This assumption is quite reasonable as a decoder has n inputs and as many as 2^n outputs. No other restriction is imposed on its implementation.

Figure A.1 shows a schematic diagram of a decoder which has k primary inputs labeled $x_1, x_2, \ldots, x_k$, and n primary outputs labeled $c_1, c_2, \ldots, c_n$, where $n \leq 2^k$. The set of valid input vectors is a subset of the set of all possible input vectors. Therefore, the set of valid input vectors contains $n \leq 2^k$ input vectors. The set of valid input vectors which activate output c_i is denoted by $X(c_i)$. Since for any given valid input vector one and only one output is activated $|X(c_i)| = 1$ and $X(c_i) \cap X(c_j) = \emptyset$ if and only if $c_i \neq c_j$.

Figure A.1 also shows the last level of gates just before the primary outputs. In order to maintain complete generality, each gate is shown as a module and not as a specific gate (such as AND, OR, NOR or NAND). These gates are labeled with the corresponding outputs. c_i becomes active if and only if all inputs to gate c_i are active. (For AND and NAND gates, logic 1 is an active input; while for OR and NOR gates, logic 0 is an active input.) If the output of gate c_i is active, the output of gate c_j must be inactive ($c_i \neq c_j$). Thus when c_i is active, at least one input of gate c_j must remain inactive to ensure that c_j is inactive. We can partition the inputs of gate c_j into two sets, $A(c_j^i)$ and $I(c_j^i)$, where $A(c_j^i)$ is the set of inputs of gate c_j which are active

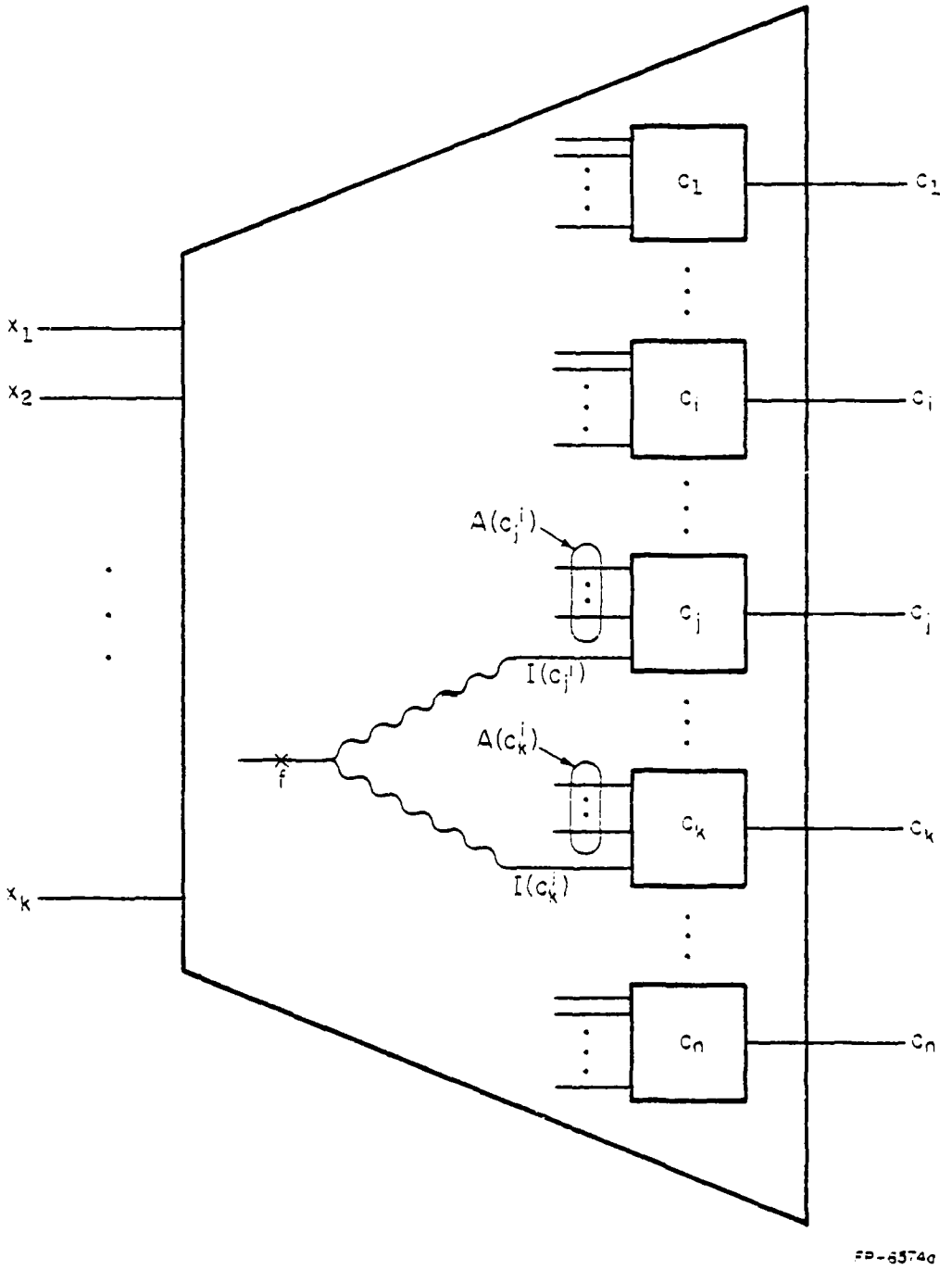


Figure A.1. Schematic diagram of a decoder illustrating the notation used in the proof of Theorem 3.1.

when output c_i is active, and $I(c_j^i)$ is the set of inputs of gate c_j which are inactive when output c_i is active. Note that $I(c_j^i) \neq \emptyset$. We now prove Theorem 3.1 which is restated below for easy reference.

Theorem 3.1: If a decoder is realized without any reconvergent fanout then under a single stuck-at fault its behavior can be formulated independent of its implementation detail as follows: for a given valid input to the decoder, instead of, or in addition to the expected output some other output is activated, or no output is activated.

Proof: We prove the theorem by contradiction. Assume that under a single stuck-at fault, the input vector $X(c_i)$ activates outputs c_j and c_k, in addition to, or instead of c_i. Therefore under the fault, both $I(c_j^i)$ and $I(c_k^i)$ become active in addition to $A(c_j^i)$ and $A(c_k^i)$. Since there is only a single fault, the inputs in $I(c_j^i)$ and $I(c_k^i)$ can be traced back to a line where the fault occurs. This line is denoted by f in Figure A.1. (This could be a primary input line.) Since the decoder does not have any reconvergent fanout, $|I(c_j^i)| = 1$ and $|I(c_k^i)| = 1$; moreover, the primary inputs which can be traced back from the inputs in $A(c_j^i)$ are different from those which can be traced back from line f. Similarly, the primary inputs which can be traced back from the inputs in $A(c_k^i)$ are different from those which can be traced back from line f. Thus, if there is no fault in the decoder, the logic value on line f can be controlled by changing the logic values only on those primary inputs which can be traced back from line f, without changing the logic values on inputs in $A(c_j^i)$ and $A(c_k^i)$.

We now consider the following "thought experiment" under the fault free condition.

1. Apply $X(c_i)$. This will also activate the inputs in $A(c_j^i)$ and $A(c_k^i)$.

2. If necessary, change the logic value on those primary inputs which can be traced back from line f in order to activate line f, without making the inputs in $A(c_j^i)$ or $A(c_k^i)$ inactive.

This will make both $I(c_j^i)$ and $I(c_k^i)$ active which means that both c_j and c_k will become active. Thus even though there is no fault in the decoder, some valid input vector activates both c_j and c_k, which is impossible. □

Theorem A.1: If a decoder is realized without any reconvergent fanout then under a single stuck-at fault if $X(c_i)$ activates c_j instead of, or in addition to c_i, $X(c_j)$ will activate only c_j.

Proof: We prove this theorem also by contradiction.

1) First assume that under a single stuck-at fault $X(c_i)$ activates c_j, instead of, or in addition to c_i, and $X(c_j)$ does not activate any output. Therefore the inputs in $I(c_j^i)$ can be traced back to a line where the fault occurs. This line is denoted by f in Figure A.2. Since the decoder does not have any reconvergent fanout, $|I(c_j^i)| = 1$, and no input in $A(c_j^i)$ can be traced back to line f. When $X(c_j)$ is applied no output is activated; in particular c_j is not activated. This can happen only if some input in $A(c_j^i)$ can be traced back to another fault which keeps that input permanently inactive; but this would violate the assumption of a single stuck-at fault.

2) Now assume that under a single stuck-at fault $X(c_i)$ activates c_j, instead of, or in addition to c_i, and $X(c_j)$ activates c_k, instead of, or in addition to c_j. Therefore, the inputs in $I(c_j^i)$ and $I(c_k^j)$ can be

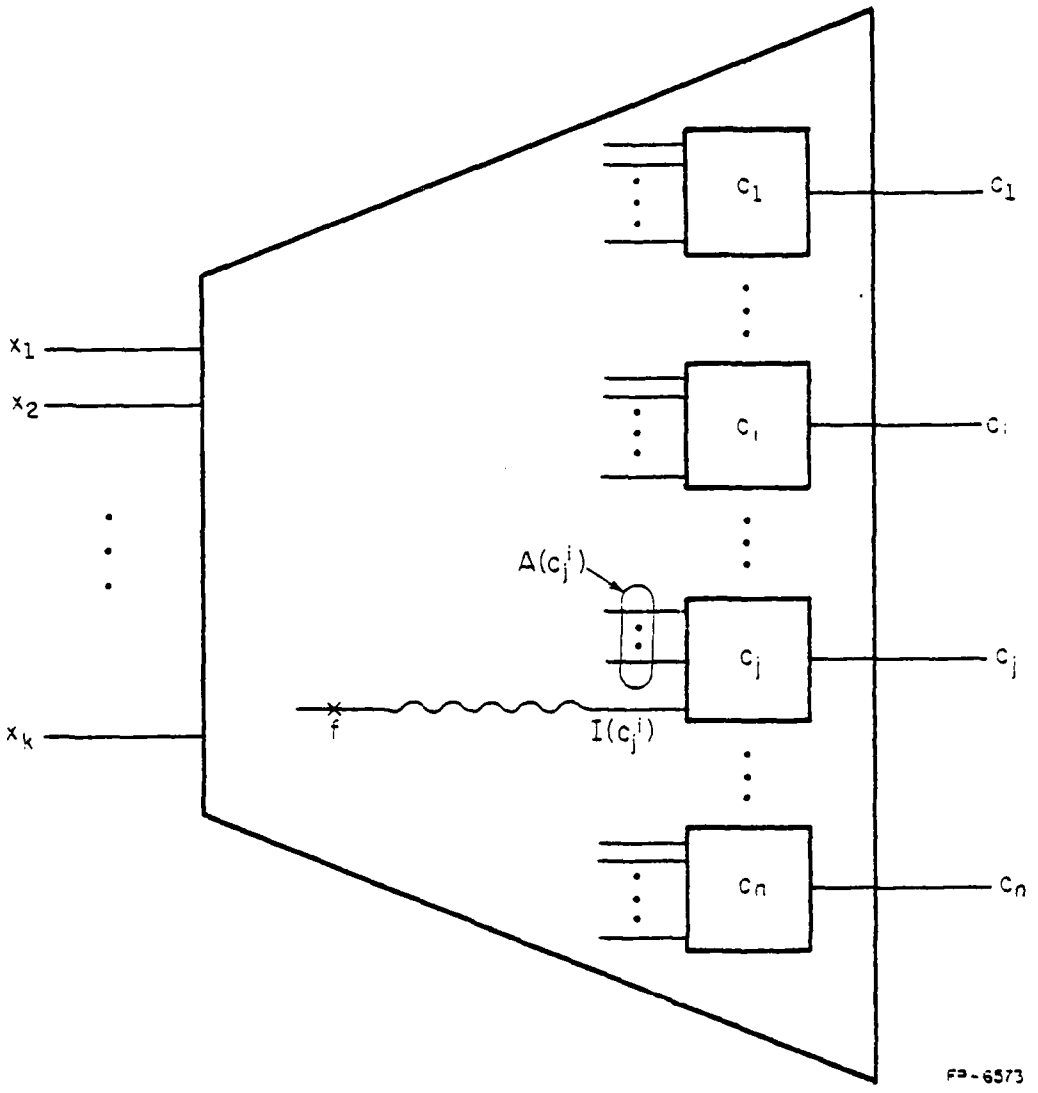


Figure A.2. Schematic diagram of a decoder illustrating the notation used in the first part of the proof of Theorem A.1.

traced back to a line where the fault occurs. This line is denoted by f in Figure A.3. Since the decoder does not have any reconvergent fanout, $|I(c_j^i)| = 1$ and $|I(c_k^j)| = 1$; moreover, the primary inputs which can be traced back from the inputs in $A(c_j^i)$ are different from those which can be traced back from line f. Similarly, the primary inputs which can be traced back from the inputs in $A(c_k^j)$ are different from those which can traced back from line f. Thus the logic value on line f can be controlled by changing the logic values only on those primary inputs which can be traced back from line f, without changing the logic values on inputs in $A(c_j^i)$ and $A(c_k^j)$.

We now consider the following "thought experiment" when there is a fault on line f as shown in Figure A.3.

1. Apply $X(c_i)$. This will also activate the inputs in $A(c_j^i)$. Due to the fault on line f, the input in $I(c_j^i)$ also becomes active, consequently activating c_j. At this time some input(s) in $A(c_k^j)$ must be inactive because c_k is not active.

2. Change the logic value on those primary inputs which can be traced back from line f in order to activate line f (for this $X(c_j)$ needs to be applied), without changing the logic values in $A(c_j^i)$ and $A(c_k^j)$, i.e., the inputs in $A(c_j^i)$ are active and some input(s) in $A(c_k^j)$ are inactive.

Thus we get in a situation where $X(c_j)$ does not activate c_k even though fault on line f exists, contradicting our assumption. □

Corollarly A.1: If a decoder is realized without any reconvergent fanout then under a single stuck-at fault if $X(c_j)$ does not activate any output, or activates c_k, instead of, or in addition to c_j, no $X(c_q)$ will activate c_j, instead of, or in addition to c_q, for $c_q \neq c_j$.

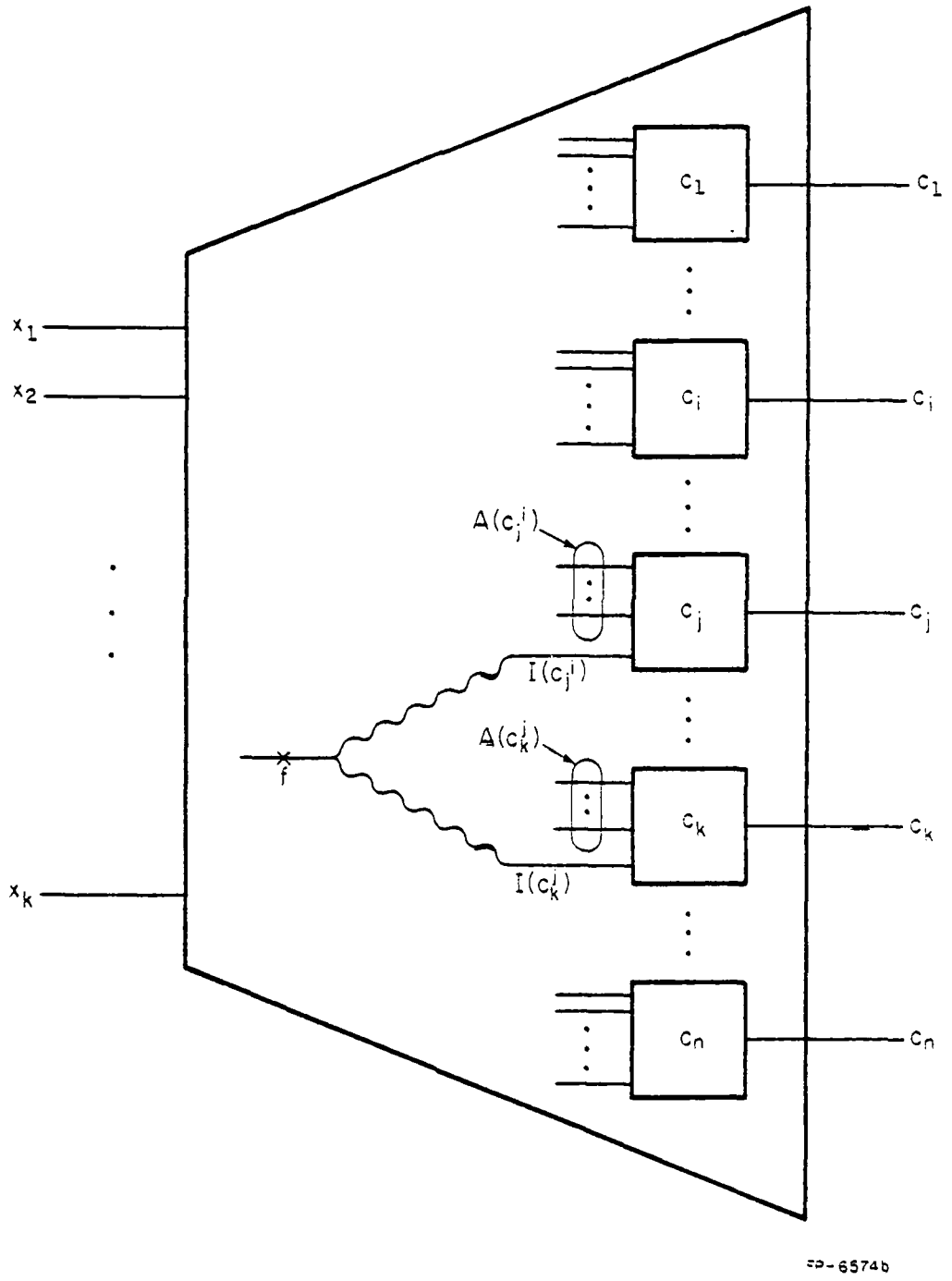


Figure A.3. Schematic diagram of a decoder illustrating the notation used in the second part of the proof of Theorem A.1.

Proof: Follows directly from Theorem A.1. □

Constraints 4 and 5 given in Section 3.2 are consistent with Theorem A.1 and Corollary A.1.

VITA

Satish Mukund Thatte was born in Poona, India on April 17, 1953. He received the B.E. (Hons.) degree in Electronics Engineering from the Birla Institute of Technology and Science, Pilani, India in 1975. At the Birla Institute of Technology and Science he received the Gold Medal for the best academic record in all branches of engineering in the graduating class of 1975. In 1977 he received the M.S. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign. He was a graduate teaching assistant in the Department of Electrical Engineering from August 1975 to December 1975 and a graduate research assistant with the Fault-Tolerant Systems and Computer Architecture group at the Coordinated Science Laboratory from January 1976 to May 1979. He is listed in the 1979 Edition of Who's Who in Technology Today.